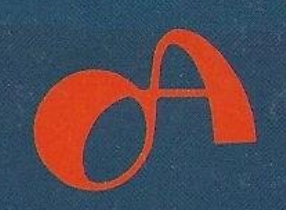


6502

PROGRAMMIEREN IN ASSEMBLER

LANCE A. LEVENTHAL

6502

PROGRAMMIEREN IN ASSEMBLER

LANCE A. LEVENTHAL

te-wi

VORWORT

Das vorliegende Buch enthält eine ausführliche Beschreibung der Assemblersprache des Mikroprozessors 6502. Es enthält zahlreiche Programmierbeispiele, vom einfachen Laden eines Speichers bis zu vollständigen Entwicklungsprojekten.

Das Schwergewicht liegt auf einer großen Anzahl von völlig fehlerfreien, praktischen Programmierbeispielen im Standard-Format, einschließlich Flußdiagramm, Quellprogramm, Objektcode und erläuterndem Text.

Ferner ist jeder Befehl des 6502 detailliert erklärt, wie auch die Assembler-Vereinbarungen für den 6502. Ebenso werden Eingabe/Ausgabe-Operationen mit dem peripheren Interface-Adapter 6520 (PIA), dem Versatile-Interface-Adapter 6522 (VIA), den Mehrfunktions-Bausteinen 6530 und 6532 sowie dem asynchronen Kommunikations-Interface-Baustein 6850 behandelt.

Ausführliche Besprechungen für die Erstellung von Programmen, von der Definition der Aufgabe über Testen, Fehlersuche, Dokumentation bis hin zu modularer und strukturierter Programmierung runden dieses Werk ab.

IMPRESSUM

Dieses Buch ist eine Übersetzung der amerikanischen Originalausgabe "6502 ASSEMBLY LANGUAGE PROGRAMMING" by Lance A. Leventhal.

LIZENZNEHMER und HERAUSGEBER:
te-wi Verlag GmbH, Theo-Prosel-Weg 1
8000 München 40

GESAMTHERSTELLUNG:
technik marketing, München
tm 3366/485, 4. Auflage
Printed in Austria

ISBN 3-921803-10-1

INHALTSVERZEICHNIS

Seite

KAPITEL 1
EINFÜHRUNG IN DIE ASSEMBLER-PROGRAMMIERUNG 1-1
Wie dieses Buch gedruckt ist 1-1
Die Bedeutung von Befehlen 1-1
Ein Computerprogramm 1-2
Die Probleme des Programmierens 1-2
Verwendung von Oktal- oder Hexadezimalzahlen 1-3
Mnemoniks für Befehlscodes 1-5
Das Assemblerprogramm 1-5
Zusätzliche Eigenschaften von Assemblern 1-7
Nachteile der Assemblersprache 1-7
Höhere Programmiersprachen 1-8
Vorteile von höheren Programmiersprachen 1-9
Nachteile von höheren Programmiersprachen 1-9
Höhere Sprachen für Mikroprozessoren 1-11
Welches Niveau sollte man verwenden? 1-13
Wie sieht es mit der Zukunft aus 1-14
Weshalb dieses Buch? 1-14
Literatur 1-15

KAPITEL 2
ASSEMBLER 2-1
Eigenschaften von Assemblern 2-1
Assembler-Befehle 2-1
Marken (Labels) 2-3
Assembler-Operationscodes (Mnemoniks) 2-5
Pseudo-Operationen 2-6
Die Pseudo-Operation DATA 2-7
Die Pseudo-Operation EQUATE (oder DEFINE) 2-8
Die Pseudo-Operation ORIGIN 2-9
Die Pseudo-Operation RESERVE 2-10
Binde-Pseudo-Operationen 2-11
Haushaltungs-Pseudo-Operationen 2-12
Markierungen bei Pseudo-Operationen 2-12
Adressen und das Operandenfeld 2-13
Bedingte Assemblierung 2-15
Makros 2-16
Kommentare 2-18
Assembler-Typen 2-19
Fehler 2-20
Lader 2-21
Literatur 2-22

KAPITEL 3

DER BEFEHLSSATZ DES 6502 IN ASSEMBLERSPRACHE 3-1
CPU-Register und Statusflags 3-4
Speicher-Adressierungsarten des 6502 3-6
Speicher - Unmittelbar 3-7
Speicher - Direkt 3-8
Implizierte oder inhärente Adressierung 3-9
Akkumulator-Adressierung 3-9
Vor-indizierte indirekte Adressierung 3-10
Nach-indizierte indirekte Adressierung 3-11
Indizierte Adressierung 3-12
Indirekte Adressierung 3-14
Relative Adressierung 3-15
Befehlssatz des 6502 3-16
Abkürzungen 3-16
Befehls-Mnemoniks 3-18
Befehls-Objektcodes 3-18
Befehls-Ausführungszeiten 3-18
Status 3-19
ADC – Add Memory, with Carry, to Accumulator 3-39
AND – AND Memory with Accumulator 3-41
ASL – Shift Accumulator or Memory Byte Left 3-43
BCC – Branch if Carry Clear (C = 0) 3-45
BCS – Branch if Carry Set (C = 1) 3-46
BEQ – Branch if Equal to Zero (Z = 1) 3-46
BIT – Bit Test 3-47
BMI – Branch if Minus (N = 1) 3-49
BNE – Branch if Not Equal to Zero (Z = 0) 3-49
BPL – Branch if Plus (N = 0) 3-50
BRK – Force Break (Trap or Software Interrupt) 3-51
BVC – Branch if Overflow Clear (V = 0) 3-53
BVS – Branch if Overflow Set (V = 1) 3-53
CLC – Clear Carry 3-54
CLD – Clear Decimal Mode 3-55
CLI – Clear Interrupt Mask (Enable Interrupts) 3-56
CLV – Clear Overflow 3-57
CMP – Compare Memory with Accumulator 3-58
CPX – Compare Index Register with Memory 3-60
CPY – Compare Index Register Y with Memory 3-61
DEC – Decrement Memory (by 1) 3-62
DEX – Decrement Index Register X (by 1) 3-64
DEY – Decrement Index Register Y (by 1) 3-65
EOR – Exclusive-OR Accumulator with Memory 3-66
INC – Increment Memory (by 1) 3-68
INX – Increment Index Register X (by 1) 3-70
INY – Increment Index Register Y (by 1) 3-71
JMP – Jump via Absolute or Indirect Addressing 3-72
JSR – Jump to Subroutine 3-74
LDA – Load Accumulator from Memory 3-75
LDX – Load Index Register X from Memory 3-77
LDY – Load Index Register Y from Memory 3-79
LSR – Logical Shift Right of Accumulator or Memory 3-81
NOP – No Operation 3-84
ORA – Logically OR Memory with Accumulator 3-85
PHA – Push Accumulator onto Stack 3-87
PHP – Push Status Register (P) onto Stack 3-88
PLA – Pull Contents of Accumulator from Stack 3-89
PLP – Pull Contents of Status Register (P) from Stack 3-90

ROL – Rotate Accumulator or Memory Left through Carry 3-91
ROR – Rotate Accumulator or Memory Right, through Carry 3-94
RTI – Return from Interrupt 3-97
RTS – Return from Subroutine 3-99
SBC – Subtract Memory from Accumulator with Borrow 3-100
SEC – Set Carry 3-102
SED – Set Decimal Mode 3-103
SEI – Set Interrupt Mask (Disable Interrupts) 3-104
STA – Store Accumulator in Memory 3-105
STX – Store Index Register X in Memory 3-106
STY – Store Index Register Y in Memory 3-108
TAX – Move from Accumulator to Index Register X 3-109
TAY – Move from Accumulator to Index Register Y 3-110
TSX – Move from Stack Pointer to Index Register X 3-111
TXA – Move from Index Register X to Accumulator 3-112
TXS – Move from Index Register X to Stack Pointer 3-113
TYA – Move from Index Register Y to Accumulator 3-114
Kompatibilität 6800/6502 3-115
Assembler-Vereinbarungen für den MOS-Technology 6502 3-119
Feld-Aufbau des Assemblers 3-119
Markierungen (Labels) 3-119
Pseudo-Operationen 3-120
Beispiele 3-120
Beispiele 3-121
Markierungen bei Pseudo-Operationen 3-122
Adressen 3-122
Andere Assembler-Eigenschaften 3-123

KAPITEL 4

EINFACHE PROGRAMME 4-1
Allgemeine Form der Beispiele 4-1
Richtlinien zur Lösung von Aufgaben 4-2
Programm-Beispiele 4-4
8-Bit-Datentransfer 4-4
8-Bit-Addition 4-5
Verschieben um ein Bit nach links 4-6
Ausmaskieren der höchstwertigen vier Bits 4-7
Löschen eines Speicherplatzes 4-8
Zerlegen eines Wortes 4-9
Finden der größeren von zwei Zahlen 4-10
16-Bit-Addition 4-12
Tabelle der Quadrate 4-14
Einer-Komplement 4-17
Aufgaben 4-18
16-Bit-Datentransfer 4-18
8-Bit-Subtraktion 4-18
Verschiebe um zwei Bits nach links 4-18
Maskiere die niedrigstwertigen vier Bits aus 4-18
Setze einen Speicherplatz auf Einsen 4-18
Wort-Zusammensetzung 4-19
Auffinden der kleineren von zwei Zahlen 4-19
24-Bit-Addition 4-19
Summe der Quadrate 4-20
Zweierkomplement 4-20

KAPITEL 5
EINFACHE PROGRAMMSCHLEIFEN 5-1
- Beispiele 5-4
  - Summe von Daten 5-4
  - 16-Bit-Summe von Daten 5-9
  - Anzahl der negativen Elemente 5-12
  - Finden des Maximums 5-14
- Justieren einer gebrochenen Binärzahl 5-18
  - Nach-indizierte (indirekte) Adressierung 5-21
  - Vor-indizierte (indirekte) Adressierung 5-23
- Aufgaben 5-24
  - Prüfsumme von Daten 5-24
  - Summe von 16-Bit-Daten 5-24
  - Anzahl der Nullen, positiven und negativen Zahlen 5-25
  - Finden des Minimums 5-25
  - Zähle 1-Bits 5-25

KAPITEL 6
ZEICHEN-CODIERTE DATEN 6-1
- Beispiele 6-3
  - Länge einer Zeichenreihe 6-3
  - Suchen des ersten Nicht-Zwischenraum-Zeichens 6-7
  - Ersetzen führender Nullen durch Zwischenräume 6-10
  - Addition gerader Parität zu ASCII-Zeichen 6-12
  - Übereinstimmung von Bitmustern 6-15
- Aufgaben 6-20
  - Länge einer Fernschreib-Nachricht 6-20
  - Suchen des letzten Nicht-Zwischenraum-Zeichens 6-20
  - Abrunden von dezimalen Reihen in ganzzahlige Form 6-21
  - Prüfen auf gerade Parität in ASCII-Zeichen 6-21
  - Vergleich von Reihen 6-22

KAPITEL 7
CODE-UMWANDLUNG 7-1
- Beispiele 7-2
  - Hexadezimal in ASCII 7-2
  - Dezimal in Sieben-Segment 7-4
  - ASCII in dezimal 7-9
  - BCD in binär 7-11
  - Umwandlung einer Binärzahl in eine ASCII-Reihe 7-13
- Aufgaben 7-15
  - ASCII in hexadezimal 7-15
  - Sieben-Segment in dezimal 7-15
  - Dezimal in ASCII 7-15
  - Binär in BCD 7-16
  - Binärzahl in ASCII-Reihe 7-16
- Literatur 7-17

KAPITEL 8
ARITHMETISCHE AUFGABEN 8-1
- Beispiele 8-2
  - Addition mit mehrfacher Genauigkeit 8-2
  - Dezimale Addition 8-5
  - 8-Bit-Binärmultiplikation 8-8
  - 8-Bit-Binär-Division 8-14
  - Selbst-prüfende Zahlen (Double Add Double Mod 10) 8-19
- Aufgaben 8-26
  - Binäre Subtraktion mit mehrfacher Genauigkeit 8-26
  - Dezimale Subtraktion 8-26
  - Binär-Multiplikation 8 Bit mal 16 Bit 8-27
  - Binäre Division mit Vorzeichen 8-28
  - Selbstprüfende Zahlen (Aligned 1, 3, 7 Mod 10) 8-28
- Literatur 8-30

KAPITEL 9
TABELLEN UND LISTEN 9-1
- Beispiele 9-2
  - Hinzufügen einer Eingabe zu einer Liste 9-2
  - Prüfen einer geordneten Liste 9-5
  - Entfernen eines Elementes von einer Warteschlange 9-8
  - 8-Bit-Sortierung 9-11
  - Verwendung einer geordneten Sprungtabelle 9-16
- Aufgaben 9-18
  - Entfernen einer Eingabe aus einer Liste 9-18
  - Hinzufügen einer Eingabe zu einer geordneten Liste 9-19
  - Addieren eines Elementes zu einer Warteschlange 9-20
  - 16-Bit-Sortierung 9-21
  - Verwendung einer Sprungtabelle mit einem Schlüssel 9-21
- Literatur 9-22

KAPITEL 10
UNTERPROGRAMME 10-1
- Unterprogramm-Dokumentation 10-3
- Beispiele 10-4
  - Hexadezimal in ASCII 10-5
  - Länge einer Zeichenreihe 10-8
  - Maximalwert 10-13
  - Übereinstimmung von Mustern 10-18
  - Addition mit mehrfacher Genauigkeit 10-24
- Aufgaben 10-30
  - ASCII in hexadezimal 10-30
  - Länge einer Fernschreib-Nachricht 10-30
  - Minimalwert 10-31
  - Vergleich von Reihen 10-31
  - Dezimale Subtraktion 10-33
- Literatur 10-34

KAPITEL 11

EINGABE/AUSGABE 11-1
- Zeit-Intervalle (Verzögerungen) 11-10
- Verzögerungs-Routinen 11-11
- Verzögerungsprogramm 11-12
- Eingabe/Ausgabe-Chips des 6502 11-14
- Der periphere Interface-Adapter (PIA) 6520 11-15
- PIA-Steuer-Register 11-17
- Konfigurieren des PIA 11-20
- Beispiele für die PIA-Konfiguration 11-22
- Verwendung des PIA zum Transferieren von Daten 11-25
- Der Versatile Interface-Adapter (VIA) 6522 11-27
- Konfigurieren des VIA 11-32
- CA2-Eingang 11-36
- CA2-Ausgang 11-36
- Beispiele für VIA-Konfiguration 11-38
- Verwendung des VIA zum Transfer von Daten 11-40
- Unterbrechungsflag-Register des VIA 11-41
- VIA-Zeitgeber 11-42
- Arbeitsweise des Zeitgebers 2 im VIA 6522 11-43
- Arbeitsweise des Zeitgebers 1 im VIA 6522 11-45
- Die Mehrfunktions-Bausteine 6530 und 6532 11-47
- Beispiele 11-51
  - Eine Taste 11-51
  - Ein Kippschalter 11-58
  - Ein Schalter mit mehreren Stellungen 11-63
  - Eine einzelne LED 11-72
  - Sieben-Segment-LED-Anzeige 11-76
- Aufgaben 11-87
  - Eine Ein-Aus-Taste 11-87
  - Entprellen eines Schalters durch Software 11-87
  - Steuerung für einen Drehschalter 11-87
  - Darstellung der Schalterpositionen auf Lampen 11-88
  - Zählung auf einer Sieben-Segment-Anzeige 11-88
  - Komplexere E/A-Bausteine 11-89
- Beispiele 11-93
  - Eine nicht-codierte Tastatur 11-93
  - Eine codierte Tastatur 11-102
  - Ein Digital-Analog-Wandler 11-105
  - Analog-Digital-Wandler 11-110
  - Ein Fernschreiber (TTY) 11-115
  - Der asynchrone Kommunikations-Interface-Adapter (ACIA) 6850 11-125
  - Der asynchrone Kommunikations-Interface-Adapter (ACIA) 6551 11-132
  - Logische und physikalische Bausteine 11-137
  - Standard-Interfaces 11-139
- Aufgaben 11-140
  - Trennung von Tastenbetätigung von einer nicht-codierten Tastatur 11-140
  - Lesen eines Satzes von einer codierten Tastatur 11-140
  - Ein Rechteck-Generator mit variabler Amplitude 11-141
  - Mittelwert-Bildung von analogen Ablesungen 11-141
  - Ein Terminal mit 30 Zeichen pro Sekunde 11-141
- Literatur 11-142

KAPITEL 12

UNTERBRECHUNGEN 12-1
- 6502-Unterbrechungs-System 12-4
  - 6520-PIA-Unterbrechungen 12-6
  - 6522-VIA-Unterbrechungen 12-7
  - Unterbrechungen mit Mehrfunktions-Bausteinen 6530 und 6532 12-10
  - ACIA-Unterbrechungen 12-11
  - 6502-Abfrage-Unterbrechungssysteme 12-12
  - Vektorisierte Unterbrechungs-Systeme mit dem 6502 12-13
- Beispiele 12-14
  - Eine Start-Unterbrechung 12-14
  - Eine Tastatur-Unterbrechung 12-18
  - Eine Drucker-Unterbrechung 12-23
  - Eine Echtzeit-Takt-Unterbrechung 12-26
  - Eine Fernschreiber-Unterbrechung 12-35
- Allgemeinere Service-Routinen 12-40
- Aufgaben 12-41
  - Eine Test-Unterbrechung 12-41
  - Eine Tastatur-Unterbrechung 12-41
  - Eine Drucker-Unterbrechung 12-41
  - Eine Echtzeit-Takt-Unterbrechung 12-41
  - Eine Fernschreiber-Unterbrechung 12-41
- Literatur 12-42

KAPITEL 13

AUFGABEN-DEFINITION UND PROGRAMM-ENTWICKLUNG 13-1
- Die Aufgaben der Software-Entwicklung 13-1
- Definition der Stufen 13-4
- Aufgaben-Definition 13-5
- Definition der Eingaben 13-5
- Definition der Ausgangssignale 13-6
- Verarbeitungs-Abschnitt 13-6
- Verhalten bei Fehlern 13-7
- Menschliche Faktoren 13-8
- Beispiele 13-9
  - Reaktion auf einen Schalter 13-9
  - Ein Speicher-Lader mit Schaltern 13-11
  - Ein Verifikations-Terminal 13-15
- Überblick über die Definition der Aufgabe 13-20
- Programm-Entwicklung 13-21
- Aufstellen von Flußdiagrammen 13-22
- Beispiele 13-24
  - Reaktion auf einen Schalter 13-24
  - Der Speicher-Lader mit Schaltern 13-26
  - Kreditkarten-Terminal 13-28
- Modulare Programmierung 13-33
- Beispiele 13-35
  - Reaktion auf einen Schalter 13-35
  - Der Speicher-Lader mit Schaltern 13-35
  - Das Verifikations-Terminal 13-36
- Überblick über modulare Programmierung 13-38
- Strukturierte Programmierung 13-38
- Beispiele 13-45
  - Reaktion auf einen Schalter 13-45
  - Der Speicher-Lader mit Schaltern 13-46
  - Das Kredit-Verifikations-Terminal 13-47
- Überblick über die strukturierte Programmierung 13-51

"Top-Down"-Entwicklung (Schrittweise Verfeinerung) 13-52
Beispiele 13-54
Reaktion auf einen Schalter 13-54
Der Speicher-Lader mit Schaltern 13-55
Das Verifikations-Terminal 13-57
Überblick über das Verfahren der schrittweisen Verfeinerung 13-60
Überblick über die Aufgaben-Definition und Programm-Entwicklung 13-61
Literatur 13-62

KAPITEL 14
FEHLERSUCHE UND TESTEN 14-1
Einfache Fehlersuch-Hilfsmittel 14-1
Fortschrittlichere-Fehlersuch-Hilfsmittel 14-9
Fehlersuche mit Prüflisten 14-12
Suchen von Fehlern 14-14
Fehlersuch-Beispiel 1: Dezimal/Sieben-Segment-Umwandlung 14-19
Fehlersuch-Beispiel 2: Sortieren in absteigender Reihenfolge 14-23
Einführung in das Testen 14-31
Auswahl von Testdaten 14-33
Testbeispiel 1: Sortierprogramm 14-34
Testbeispiel 2: Selbst-prüfende Zahlen 14-34
Vorsichtsmaßnahmen beim Testen 14-34
Schlußfolgerung 14-35
Literatur 14-36

KAPITEL 15
DOKUMENTATION UND NEU-ENTWICKLUNG 15-1
Selbst-dokumentierende Programme 15-1
Kommentare 15-3
Kommentier-Beispiel 1: Addition mit mehrfacher Genauigkeit 15-5
Kommentier-Beispiel 2: Fernschreiber-Ausgabe 15-7
Flußdiagramme als Dokumentation 15-9
Strukturierte Programme als Dokumentation 15-9
Speicherpläne 15-10
Parameter- und Definitionslisten 15-11
Bibliotheks-Routinen 15-13
Bibliotheks-Beispiele 15-14
Bibliotheks-Beispiel 1: Summe von Daten 15-14
Bibliotheks-Beispiel 2: Dezimal-in-Sieben-Segment-Umwandlung 15-15
Bibliotheks-Beispiel 3: Dezimale Summe 15-16
Vollständige Dokumentation 15-17
Neu-Entwicklung 15-19
Reorganisation für weniger Speicher 15-20
Größere Reorganisation 15-23
Literatur 15-25

KAPITEL 16
BEISPIELE 16-1
Projekt-Nr. 1: Eine digitale Stoppuhr 16-1
Projekt-Nr. 2: Ein Digital-Thermometer 16-17
Literatur 16-31

Kapitel 1
EINFÜHRUNG IN DIE ASSEMBLER-PROGRAMMIERUNG

Dieses Buch beschreibt die Programmierung in Assemblersprache. Es wird angenommen, daß Sie mit dem Buch "Einführung in die Mikrocomputer-Technik" (insbesondere Kapitel 6) vertraut sind. Im vorliegenden Buch werden die allgemeinen Eigenschaften von Computern, Mikrocomputern, Adressierverfahren oder Befehlssätzen nicht besprochen. Alle diese Fragen wurden in der bereits erwähnten "Einführung in die Mikrocomputer-Technik" erörtert.

WIE DIESES BUCH GEDRUCKT IST

Es ist zu beachten, daß dieses Buch mit fetten und mageren Buchstaben **gedruckt ist. Dies hilft jene Teile des Buches zu überspringen, deren Inhalt vielleicht bekannt ist. Man kann sicher sein, daß der mager gedruckte Teil nur jene Informationen ausführlich behandelt, die im vorausgehenden fettgedruckten Teil bereits dargestellt wurden.** Man braucht daher nur den fettgedruckten Teil zu lesen, bis man ein Thema erreicht, über das man mehr zu wissen wünscht und kann von hier ab mit dem Lesen des mager gedruckten Teiles beginnen.

DIE BEDEUTUNG VON BEFEHLEN

Der Befehlssatz eines Mikroprozessors besteht aus einem Satz von binären Eingangssignalen, mit denen bestimmte Tätigkeiten während eines Befehlszyklus bewirkt werden. Ein Befehlssatz ist für einen Mikroprozessor das gleiche wie eine Funktionstabelle für einen Logik-Baustein, wie etwa ein Gatter, Addierer oder Schieberegister. Natürlich sind die Tätigkeiten, die ein Mikroprozessor infolge eines Befehles ausführt, wesentlich komplexer als jene Tätigkeiten, die ein Logik-Baustein infolge seiner Eingangssignale durchführt.

BINÄRE BEFEHLE

Ein Befehl ist ein binäres Bitmuster – es muß an den Dateneingängen des Mikro-prozessors zur richtigen Zeit anliegen, um als Befehl interpretiert zu werden. Wenn beispielsweise der Mikroprozessor 6502 das aus 8 Bits bestehende Binärmuster 11101000 als Eingabe während eines Befehlsabrufes empfängt, so bedeutet dieses:

"Inkrementiere (addiere 1) zum Inhalt von Register X".

Ähnlich bedeutet das Muster 10101001:

"Lade den Akkumulator mit dem Inhalt des nächsten Wortes des Programmspeichers"

Der Mikroprozessor (wie auch jeder andere Computer) erkennt nur binäre Muster als Befehle oder Daten. Er ist nicht imstande, Worte oder Oktal-, Dezimal- oder Hexadezimalzahlen zu erkennen.

EIN COMPUTERPROGRAMM

Ein Programm besteht aus einer Serie von Befehlen, die den Computer zur Ausführung einer bestimmten Aufgabe veranlassen.

COMPUTER-PROGRAMM

In Wirklichkeit enthält ein Computerprogramm mehr als Befehle. Es enthält auch Daten und Speicheradressen, die der Mikroprozessor zur Ausführung der durch die Befehle definierten Aufgaben benötigt. Zweifellos muß der Mikroprozessor zur Ausführung einer Addition hierfür zwei Zahlen besitzen, sowie einen Platz, in den er das Resultat ablegt. Das Computerprogramm muß die Quellen der Daten und den Bestimmungsort für das Resultat bestimmen und muß ferner die auszuführende Operation angeben.

Die meisten Mikroprozessoren führen Befehle sequentiell aus, es sei denn, einer der Befehle ändert die Befehlssequenz oder hält den Computer an. Das heißt, der Prozessor erhält den nächsten Befehl von der nächsthöheren Speicheradresse, außer der momentane Befehl weist ihn an, etwas anderes auszuführen.

Jedes Programm wird letztlich in einen Satz von Binärzahlen umgewandelt. Folgendes ist beispielsweise ein Programm für den 6502, das den Inhalt der Speicherplätze 0060_{16} und 0061_{16} addiert und das Ergebnis in den Speicherplatz 0062_{16} ablegt:

```
10100101
01100000
01100101
01100001
10000101
01100010
```

OBJEKT-PROGRAMM

MASCHINEN-SPRACHEN-PROGRAMM

Dies ist ein Maschinensprachen- oder Objekt-Programm. Wenn dieses Programm in einen auf einem 6502 basierenden Mikrocomputer eingegeben wird, so ist der Mikrocomputer imstande, dieses Programm direkt auszuführen.

DIE PROBLEME DES PROGRAMIERENS

Es gibt verschiedene Schwierigkeiten, die mit der Bildung von Programmen in Form eines Objektprogrammes, d.h. eines Programmes in binärer Maschinensprache, verbunden sind. Einige dieser Probleme sind:

1) Die Programme sind schwierig zu verstehen oder fehlerfrei zu machen (Binärzahlen sehen einander sehr ähnlich, insbesondere wenn man bereits einige Stunden mit ihnen gearbeitet hat).
2) Die Eingabe der Programme ist sehr langwierig, da man jedes Bit individuell bestimmen muß.
3) Die Programme beschreiben die Aufgabe , die der Computer ausführen soll, nicht in irgendeinem vergleichbaren vom Menschen lesbaren Format.
4) Die Programme sind lang und mühsam zu schreiben.
5) Der Programmierer macht häufig Fehler, die sehr schwer zu finden sind.

Die folgende Version unseres Additions-Objektprogrammes enthält beispielsweise ein einziges vertauschtes Bit. Versuchen Sie es zu finden:

```
10100101
01100000
01110101
01100001
10000101
01100010
```

Obwohl der Computer Binärzahlen mit Leichtigkeit handhabt, ist dies für einen Menschen sehr schwierig. Der Mensch findet Binärprogramme lang, mühsam, verwirrend und bedeutungslos. Ein Programmierer könnte sich möglicherweise einige der Binärcodes merken, aber derartige Anstrengungen sollten produktiver verwendet werden.

VERWENDUNG VON OKTAL- ODER HEXADEZIMALZAHLEN

OKTAL- ODER HEXA-DEZIMALZAHLEN

Wir können diese Situation wesentlich verbessern, indem wir die Befehle in Form von Oktal- oder Hexadezimal-, anstatt von Binärzahlen zu schreiben. Wir werden in diesem Buch Hexadezimalzahlen verwenden, da sie kürzer sind und bereits zum Standard für die Mikroprozessor-Industrie geworden sind. Tabelle 1-1 definiert die Hexadezimalziffern und ihre binären Äquivalente.**Das Programm für den 6502 zur Addition zweier Zahlen sieht nun folgendermaßen aus:**

```
A5
60
65
61
85
62
```

Man sieht, daß die hexadezimale Version wesentlich kürzer in der Schreibweise ist und lange nicht so ermüdend bei der Überprüfung.

Fehler sind wesentlich leichter in einer Folge von Hexadezimalziffern zu finden. Die fehlerhafte Version unseres Additionsprogramms sieht nun in hexadezimaler Form folgendermaßen aus:

```
A5
60
75
61
85
62
```

Der Fehler ist wesentlich einfacher zu erkennen.

Was fangen wir aber nun mit diesem hexadezimalen Programm an? Der Mikroprozessor versteht nur binäre Befehlscodes. Die Antwort ist, daß wir die Hexadezimalzahlen in Binärzahlen umwandeln müssen. Diese Umwandlung ist eine sich wiederholende mühsame Aufgabe. Bei dieser Umwandlung macht man im allgemeinen alle möglichen Arten der schönsten Fehler, wie etwa das Verwechseln einer Zeile, Vergessen eines Bits, oder Vertauschen eines Bits oder einer Stelle.

Diese sich wiederholende und zermürbende Arbeit ist jedoch eine ideale Aufgabe für einen Computer. Der Computer wird niemals müde oder gelangweilt und macht niemals leichtfertige Fehler.

HEXADEZIMAL-LADER

Es liegt daher nahe, ein Programm zu schreiben, das die hexadezimalen Zahlen aufnimmt, sie in Binärzahlen umwandelt und die Binärzahlen in den Speicher des Mikrocomputers bringt. Dies ist ein Standardprogramm, das für viele Mikroprozessoren erhältlich ist. Es wird Hexadezimal-Lader genannt.

Lohnt sich ein Hexadezimal-Lader? Wenn man die Absicht hat, ein Programm unter Verwendung von Binärzahlen zu schreiben, und wir bereit sind, das Programm in seiner binären Form in den Computer einzugeben, dann werden wir keinen Hexadezimal-Lader benötigen. Wenn wir einen Hexadezimal-Lader wählen, so müssen wir einen Preis hierfür zahlen. Der Hexadezimal-Lader ist selbst ein Programm, das man in den Speicher laden muß. Außerdem wird der Hexadezimal-Lader Speicherplatz besetzen – Speicherplatz, den wir vielleicht für andere Zwecke benötigen würden.

Die Vorteile des Hexadezimal-Laders liegen, abgesehen von den Kosten und den Speicher-Anforderungen in der Einsparung der Programmierzeit.

Ein Hexadezimal-Lader ist daher seine geringen Kosten wert.

Ein Hexadezimal-Lader löst aber keinesfalls irgendein Programmierproblem. Die hexadezimale Version des Programmes ist noch immer schwierig zu lesen und zu verstehen. Er unterscheidet beispielsweise weder Befehle von Daten oder Adressen, noch ergibt sich aus der Auflistung des Programmes irgendein Hinweis darauf, was das Programm ausführt. Was bedeutet 86 oder D0? Das Auswendiglernen einer langen Reihe von Codes ist sicherlich nicht sehr attraktiv. Ferner unterscheiden sich die Codes völlig bei unterschiedlichen Mikroprozessoren, und das Programm benötigt eine umfangreiche Dokumentation.

Tabelle 1-1. Hexadezimal-Umwandlungstabelle

Hexadezimal-Ziffer	Binäres Äquivalent	Dezimales Äquivalent
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

MNEMONIKS FÜR BEFEHLSCODES

Eine offensichtliche Verbesserung des Programmierens stellt die Zuweisung einer Bezeichnung zu jedem Befehlscode dar. Den Namen eines Befehlscodes nennt man ein "Mnemonik", oder mnemotechnisches Hilfsmittel. Die mnemotechnische Abkürzung eines Befehls sollte in irgendeiner Form beschreiben, was der Befehl ausführt.

PROBLEME MIT MNEMONIKS

In der Tat liefert jeder Hersteller eines Mikroprozessors einen Satz von Mnemoniks für den Befehlssatz seines Mikroprozessors. **Man muß sich jedoch nicht mit dem Befehlssatz des Herstellers abfinden.** Sie sind jedoch Standard für einen gegebenen Mikroprozessor und werden daher von allen Anwendern verstanden. Daher stellen sie die Bezeichnungen für die Befehle dar, die man in Handbüchern, Listen, Büchern, Artikeln und Programmen findet. Das Problem bei der Auswahl von Mnemoniks für Befehle besteht darin, daß nicht alle Befehle "offensichtliche" Namen besitzen. Bei einigen Befehlen ist dies der Fall (z.B. ADD, AND, OR), andere bestehen aus offensichtlichen Abkürzungen (z.B. SUB für Subtraktion, XOR für Exklusiv-ODER), während dies bei anderen überhaupt nicht der Fall ist. Daraus resultieren Mnemoniks wie etwa WMP, PCHL und auch SOB (versuchen Sie ihre Bedeutung zu erraten!). Die meisten Hersteller verwenden einige brauchbare Namen, zum Teil aber auch sehr unglückliche Bezeichnungen. Anwender, sie sich ihre eigenen Mnemoniks ausdenken, werden wahrscheinlich kaum bessere Bezeichnungen als der Hersteller finden.

Zusammen mit den Befehls-Mnemoniks wird ein Hersteller gewöhnlich auch den CPU-Registern entsprechende Namen zuweisen. Ebenso wie bei den Befehlsnamen besitzen einige Register offensichtliche Bezeichnungen (z.B. A für Akkumulator) während andere mehr historische Bedeutung besitzen. Wir wollen jedoch wiederum die Vorschläge der Hersteller verwenden, einfach um eine Standardisierung zu fördern.

PROGRAMM IN ASSEMBLER-SPRACHE

Wenn wir die Standard-Mnemoniks für Befehle und Register des 6502 verwenden, wie sie von der Firma MOS Technology definiert wurden, so wird unser Additionsprogramm für den 6502 folgendermaßen aussehen:

```
LDA    $60
ADC    $61
STA    $62
```

Die Darstellung des Programms ist immer noch nicht völlig offensichtlich, aber es werden wenigstens einige Teile hiervon verständlich. ADC ist eine beträchtliche Verbesserung gegenüber 65. LDA und STA lassen auf Laden (Loading) und Speichern (Storing) schließen. Wir wissen nun, welche Zeilen Befehle sind und welche Daten oder Adressen. **Ein derartiges Programm stellt ein Programm in Assemblersprache dar.**

DAS ASSEMBLERPROGRAMM

HAND-ASSEMBLIERUNG

Wie bekommen wir nun das Programm in das Programm in Assemblersprache in den Computer? Wir müssen es übersetzen, entweder in Hexadezimal- oder Binärzahlen.

Man kann ein Programm in Assemblersprache manuell übersetzen, Befehl für Befehl. Dies wird Hand-Assemblierung genannt. Hand-Assemblierung einer aus drei Befehlen bestehende Folge ist nachstehend gezeigt.

Befehls-Mnemonik	Adressier-Art	Hexadezimales Äquivalent
LDA	Null-Seite (direkt)	A5
ADC	Null-Seite (direkt)	65
STA	Null-Seite (direkt)	85

Wie im Falle der Umwandlung hexadezimal in binär, ist die Hand-Assemblierung eine Routine-Aufgabe, die uninteressant, sich wiederholend und anfällig für zahllose kleine Irrtümer ist. Verwenden der falschen Zeile, Vertauschen von Ziffern, Auslassen von Befehlen und falsches Lesen der Codes sind nur einige der Fehler, die einem unterlaufen können. Die meisten Mikroprozessoren machen die Aufgaben noch komplizierter, indem sie verschieden Befehle mit unterschiedlichen Wortlängen besitzen. Manche Befehle sind nur ein Wort lang, während andere eine Länge von zwei oder drei Worten besitzen. Manche Befehle benötigen Daten im zweiten und dritten Wort. Andere benötigen Speicheradressen, Registernummern oder andere Informationen.

ASSEMBLER

QUELLEN-PROGRAMM

OBJEKT-PROGRAMM

Die Assemblierung ist eine weiter routinemäßige Aufgabe, die wir einem Mikrocomputer überlassen können. Der Mikrocomputer macht niemals Fehler beim Übersetzen von Codes. Er weiß immer, wieviele Worte und welches Format jeder Befehl benötigt. Ein Programm, das eine derartige Aufgabe ausführt, wird Assembler genannt. Das Das Assemblerprogramm übersetzt ein Anwenderprogramm oder Quellenprogramm (auch als Quellprogramm bezeichnet), das mit Mnemoniks geschrieben wurde, in ein Programm in Maschinensprache (oder Objektprogramm), das der Mikrocomputer ausführen kann. Die Eingabe für den Assembler ist ein Quellenprogramm und er gibt ein Objektprogramm aus. (Bei der Benützung des Wortes "Assembler" ist jeweils darauf zu achten, ob nun die Assembler-Sprache oder das Assembler-Programm gemeint ist).

Die Überlegungen, die wir beim Hexadezimal-Lader angestellt haben, gelten in erhöhtem Maße für den Assembler). Assembler sind aufwendiger, benötigen mehr Speicherplatz und erfordern längere Ausführungszeiten als Hexadezimal-Lader. Während Anwender häufig ihre eigenen Lader schreiben, geschieht dies weniger häufig als bei Assemblern.

Assembler besitzen eigene Regeln, die man erlernen muß. Diese enthalten die Verwendung bestimmter Markierungen oder Kennzeichen (wie Zwischenräume, Kommata, Strichpunkte oder Doppelpunkte) an den entsprechenden Stellen, korrekte Aussprache, die richtige Steuer-Information und vielleicht auch die ordnungsgemäße Plazierung von Namen und Zahlen. Diese Regeln sind gewöhnlich einfach und können rasch erlernt werden.

ZUSÄTZLICHE EIGENSCHAFTEN VON ASSEMBLERN

Frühere Assembler-Programme leisteten wenig mehr als die Übersetzung der Mnemoniks der Befehle und Register in ihr binäres Äquivalent. Die meisten neueren Assemblerprogramme besitzen zusätzliche Eigenschaften, wie:

1) Sie gestatten dem Anwender die Zuweisung von Namen zu Speicherplätzen, Eingabe- und Ausgabe-Bausteinen und sogar vollständigen Befehls-Sequenzen.
2) Die Umwandlung von Daten oder Adressen von verschiedenen Zahlensystemen (z.B. dezimal oder hexadezimal) in binäre und die Umwandlung von Zeichen in ihre ASCII- oder EBCDIC-Binärcodes.
3) Die Ausführung einiger arithmetischer Operationen als Teil des Assemblier-Vorganges.
4) Die Mitteilung an das Ladeprogramm, wohin Teile des Programmes oder Daten in den Speicher plaziert werden sollen.
5) Sie gestatten dem Anwender die Zuweisung von Speicherbereichen für zeitweilige Datenspeicherung und die Plazierung fester Daten in bestimmte Bereiche des Programmspeichers.
6) Die Lieferung von Informationen, die für Standardprogramme aus einer Programm-Bibliothek erforderlich sind, oder für Programme die zu einer anderen Zeit geschrieben wurden, für das momentane Programm.
7) Sie gestatten dem Anwender die Steuerung des Formats der Programm-Auflistung und der verwendeten Eingabe- und Ausgabe-Bausteine.

AUSWAHL EINES ASSEMBLERS

Alle diese Eigenschaften bedingen natürlich zusätzliche Kosten und Speicher. Mikrocomputer haben im allgemeinen wesentlich einfachere Assembler als große Computer, tendieren jedoch immer größer zu werden. Man besitzt häufig eine Auswahl an verschiedenen Assembler-Programmen. Das wichtigste Kriterium hierbei ist nicht, wieviele ausgefallene Eigenschaften der Assembler besitzt, sondern wie bequem er bei der Anwendung in der Praxis ist.

NACHTEILE DER ASSEMBLERSPRACHE

Der Assembler, ebenso wie der Hexadezimal-Lader, löst keinesfalls die gesamten Probleme des Programmierens. Ein Problem besteht in der gewaltigen Lücke zwischen dem Befehlssatz des Mikrocomputers und den Aufgaben, die der Mikrocomputer auszuführen hat. Computerbefehle führen meist Dinge aus, wie das Addieren des Inhalts zweier Register, Verschieben des Inhalts des Akkumulators um ein Bit, oder das Plazieren eines neuen Wertes in den Befehlszähler. Auf der anderen Seite möchte ein Anwender eines Mikrocomputers etwa die Prüfung, ob eine analoge Ablesung einen bestimmten Wert überschritten hat, nach einem bestimmten Kommando eines Fernschreibers Ausschau halten und auf dieses zu reagieren, oder ein Relais zur richtigen Zeit aktivieren. Ein Programmierer, der die Assemblersprache verwendet, muß derartige Aufgaben in eine Folge (oder Sequenz) einfacher Computerbefehle umwandeln. Diese Umwandlung kann eine schwierige und zeitraubende Aufgabe darstellen.

Ferner muß man beim Programmieren in Assemblersprache eine sehr detaillierte Kenntnis des verwendeten speziellen Mikrocomputers besitzen. Man muß wissen, welche Register und Befehle der Mikrocomputer hat, wie die Befehle die verschiedenen Register genau beeinflussen, welche Adressier-Verfahren der Computer verwendet und eine Unmenge weiterer Informationen. Keine dieser Informationen ist für die Aufgabe, die der Mikrocomputer letztlich ausführen muß, relevant.

Programme in Assemblersprache sind nicht übertragbar. Jeder Mikrocomputer besitzt seine eigene Assemblersprache, die durch seine Architektur bestimmt wird. Ein Programm in Assemblersprache, das für den 6502 geschrieben wurde, wird nicht auf dem 6800, dem Z80, 8080 oder dem 3870 laufen. Unser Additionsprogramm würde beispielsweise für den 8080 folgendermaßen lauten:

ÜBERTRAGBARKEIT

```
LDA   60H
MOV   B,A
LDA   61H
ADD   B
STA   62H
```

Da die Programme nicht übertragbar sind, bedeutet dies, daß wir unser Programm in Assemblersprache nicht auf einem anderen Mikrocomputer verwenden können. Das bedeutet auch weiter, daß wir nicht imstande sind, irgendein Programm zu verwenden, das nicht für unseren Mikrocomputer speziell geschrieben wurde. Dies ist ein Nachteil für Mikrocomputer, da diese Bausteine neu sind und noch nicht allzuviele Programme in Assemblersprache existieren. Daraus ergibt sich häufig, daß man auf sich selbst angewiesen ist. Wenn man ein Programm für die Ausführung einer speziellen Aufgabe benötigt, wird man dies wahrscheinlich nicht in den kleinen Programm-Bibliotheken der Hersteller finden. Man wird es auch wahrscheinlich kaum in einem Archiv, Zeitungsartikel oder einer alten Programm-Bibliothek finden. Wahrscheinlich muß man sich sein Programm selbst schreiben.

HÖHERE PROGRAMMIERSPRACHEN

Die Lösung für viele der mit Assemblersprachen-Programmen verbundenen Schwierigkeiten ist die Verwendung von "höheren" oder "problem-orientierten" Sprachen. Derartige Sprachen gestatten die Beschreibung von Aufgaben in einer Art und Weise, die eher problemorientiert als computer-orientiert ist. Jede Anweisung in einer höheren Programmiersprache führt eine erkennbare Funktion aus. Sie wird im allgemeinen mehreren Befehlen in Assemblersprache entsprechen. Ein Programm, das Kompilierer (oder Compiler) genannt wird, übersetzt ein Quellenprogramm höheren Programmiersprache in Befehle in Objektcode oder Maschinensprache.

KOMPILIERER

Es existieren mehrere unterschiedliche höhere Sprachen für verschiedene Arten von Aufgaben. Wenn wir beispielsweise die Aufgabe für unseren Computer in einer algebraischen Darstellung ausdrücken können, so können wir unser Programm in FORTRAN (FORmula TRANslation language) schreiben, der ältesten und am meisten verbreiteten höheren Programmiersprache. Wenn wir nun zwei Zahlen addieren wollen, so müssen wir dem Computer nur sagen:

FORTRAN

SUM = NUMB1 + NUMB2

Dies ist um einiges einfacher (und auch kürzer) als irgendein äquivalentes Programm in Maschinensprache oder ein äquivalentes Programm in Assemblersprache. Andere höhere Programmiersprachen sind COBOL (für kaufmännische Anwendungen), ALGOL und PASCAL (andere algebraische Sprachen), PL/1 (eine Kombination von FORTRAN, ALGOL und COBOL) und APL und BASIC (Sprachen für Time-Sharing-Systeme und Hobby-Computer).

VORTEILE VON HÖHEREN PROGRAMMIERSPRACHEN

Offensichtlich erleichtern höhere Sprachen das Programmieren und beschleunigen die Erstellung eines Programmes. Man kann etwa sagen, daß ein Programmierer ein Programm zehnmal schneller in einer höheren Sprache als in der Assemblersprache schreiben kann. Dies betrifft nur das Schreiben des Programms. Es enthält nicht die Definition der Aufgabe, Entwicklung des Programms, Fehlersuche, Testen oder Dokumentation, das ebenfalls einfacher und schneller wird. Das Programm in der höheren Sprache ersetzt beispielsweise zum Teil eine entsprechende Dokumentation. Sogar wenn man FORTRAN nicht beherrscht, kann man wahrscheinlich erkennen, was die oben dargestellten Anweisungen besagen.

Höhere Sprachen lösen viele andere Probleme, die mit der Programmierung in Assemblersprache verknüpft sind. Die höhere Sprache besitzt ihre eigene Syntax (gewöhnlich definiert durch einen nationalen oder internationalen Standard). In der Sprache werden Befehlssatz, die Register oder andere Eigenschften eines speziellen Computers nicht erwähnt. Auf alle derartige Details muß der Compiler achten. Die Programmierer können sich auf ihre eigentliche Aufgabe konzentrieren. Sie brauchen die verwendete CPU-Architektur nicht im Detail zu verstehen, das heißt, sie brauchen nicht einmal etwas über den Computer zu wissen, den sie programmieren.

MASCHINENUNABHÄNGIGKEIT VON HÖHEREN PROGRAMMIERSPRACHEN

Programme, die in einer höheren Sprache geschrieben wurden, sind übertragbar; zumindest in der Theorie. Sie werden auf jedem beliebigen Computer laufen, der einen Standard-Kompilierer für diese Sprache besitzt. Dadurch sind aber alle früher in einer höheren Sprache geschriebenen Programme für bestimmte Computer auch verfügbar, wenn man einen neuen Computer programmieren will. Dies kann im Falle der gebräuchlichsten Sprachen wie FORTRAN oder BASIC bedeuten, daß man tausende von Programmen zur Verfügung hat.

ÜBERTRAGBARKEIT VON HÖHEREN SPRACHEN

NACHTEILE VON HÖHEREN PROGRAMMIERSPRACHEN

Wenn nun aber alle erwähnten Vorteile, die wir bei höheren Programmiersprachen aufgezählt haben, wahr sind, wenn man Programme schneller schreiben kann und sie außerdem übertragbar sind, warum quält man sich dann überhaupt noch mit Assembler-Sprachen? Wer würde sich noch mit Registern, Befehlscodes, Mnemoniks und all diesem Ballast herumschlagen? Wie gewöhnlich steht allen Vorteilen meist auch eine entsprechende Anzahl von Nachteilen gegenüber.

Ein offensichtliches Problem liegt darin, daß man die "Regeln" oder die "Syntax" jeder höheren Programmiersprache, die man verwenden will, lernen muß. Eine höhere Sprache besitzt einen ziemlich komplizierteren Satz von Regeln.

SYNTAX VON HÖHEREN SPRACHEN

Man wird finden, daß man viel Zeit benötigt, ein Programm zu erhalten, das syntaktisch korrekt ist (und sogar dann wird es wahrscheinlich noch nicht das tun, was man will). Eine höhere Computersprache ist wie eine Fremdsprache. Wenn man ein wenig Talent besitzt, wird man sich rasch mit den Regeln vertraut machen, und imstande sein, Programme zu liefern, die der Kompilierer annimmt. Aber mit dem Lernen der Regeln und dem Versuch, das Programm für den Kompilierer annehmbar zu machen, ist unsere Aufgabe noch lange nicht gelöst.

Einige FORTRAN-Regeln sehen etwa folgendermaßen aus:

- Marken (Labels) müssen gänzlich aus Zahlen bestehen und in die ersten fünf Kartenspalten plaziert werden
- Anweisungen müssen in Spalte sieben beginnen
- Ganzzahlige Variable müssen mit dem Buchstaben I, J, K, L, M oder N beginnen

KOSTEN VON KOMPILIERERN

Ein weiteres Problem liegt darin, daß man einen Kompilierer zur Übersetzung der in einer höheren Sprache geschriebenen Programme in Maschinensprache benötigt. Kompilierer sind aufwendig und benötigen große Speicher. Während die meisten Assembler von 2K bis 16K Bytes eines Speichers belegen (1K = 1024), benötigt ein Kompilierer gewöhnlich wesentlich mehr Speicherplatz. Daher liegen die Kosten bei der Verwendung eines Kompilierers wesentlich höher.

ALGEBRAISCHE SCHREIBWEISE

Ferner machen nicht alle Kompilierer unsere Aufgabe tatsächlich leichter. FORTRAN ist beispielsweise sehr gut für Aufgaben geeignet, die man als algebraische Formeln darstellen kann. Wenn wir jedoch die Aufgabehaben, einen Drucker zu steuern, eine Kette von Zeichen zu editieren oder ein Alarmsystem zu überwachen, kann unsere Aufgabe nicht ohne weiteres in algebraischer Schreibweise dargestellt werden. In der Tat kann die Formulierung der Lösung in algebraischer Schreibweise wesentlich mühsamer und schwieriger sein, als ihre Formulierung in Assemblersprache. Die naheliegende Antwort wäre natürlich die Verwendung einer besser geeigneten höheren Sprache. Es existieren einige derartige Sprachen, aber sie sind weit weniger verbreitet und standardisiert wie FORTRAN. Man wird nicht all zu viel von den Vorteilen höherer Sprachen haben, wenn man diese sogenannten "System-Implementation-Languages" (etwa: System-Durchführungs-Sprachen) verwendet.

INEFFIZIENZ HÖHERER SPRACHEN

OPTIMIERENDE KOMPILIERER

Höhere Sprachen ergeben häufig wenig effiziente Programme in Maschinensprache. Der wesentliche Grund hierfür ist, daß die Kompilierung einen automatischen Vorgang darstellt, der voll mit Kompromissen für die Ausführung zahlreicher Möglichkeiten ist. Der Kompilierer arbeitet ähnlich einem computergesteuerten Sprach-Übersetzer, bei dem die Worte manchmal stimmen, aber der Klang und der Satzbau wirkt äußerst unbeholfen. Ein einfacher Kompilierer kann nicht wissen, wann eine Variable nicht länger verwendet wird und gelöscht werden kann, wann ein Register besser anstelle eines Speicherplatzes verwendet werden soll, oder wann Variable einfache Beziehungen untereinander besitzen. Der erfahrene Programmierer kann die Vorteile von Abkürzungen verwenden, um die Ausführungszeit zu verringern oder die Verwendung des Speichers zu reduzieren. Einige wenige Kompilierer (bekannt als optimierende Kompilierer) können dies ebenfalls, aber derartige Kompilierer sind wesentlich größer und langsamer als gewöhnliche Kompilierer.

Die allgemeinen Vorteile und Nachteile höherer Programmiersprachen sind:

Vorteile:

VORTEILE HÖHERER SPRACHEN

- Bequemer für die Beschreibung von Aufgaben
- Größere Produktivität des Programmierers
- Einfachere Dokumentation
- Standardisierte Syntax
- Unabhängigkeit von der Struktur eines speziellen Computers
- Übertragbarkeit
- Verfügbarkeit von Bibliotheken und anderen Programmen

Nachteile:

NACHTEILE HÖHERER SPRACHEN

- Spezielle Regeln
- Weitgehende Hardware- und Software-Unterstützung erforderlich
- Orientierung der gebräuchlichen Sprachen auf algebraische oder kaufmännische Probleme
- Ineffiziente Programme
- Schwierigkeiten bei der Optimierung des Codes für zeitliche und Speicher-Anforderungen
- Unmöglichkeit der bequemen Verwendung spezieller Eigenschaften eines Computers

HÖHERE SPRACHEN FÜR MIKROPROZESSOREN

Die Anwender von Mikroprozessoren werden auf verschiedene spezielle Schwierigkeiten stoßen, wenn sie höhere Programmiersprachen verwenden. Diese sind unter anderem:

- **Es existieren wenige höhere Programmiersprachen für Mikroprozessoren.**
- **Es sind keine Standardsprachen allgemein verfügbar.**
- **Wenige Kompilierer laufen tatsächlich auf Mikrocomputern. Die es tun, benötigen oft sehr großen Speicherraum.**
- **Die meisten Mikroprozessor-Anwendungen sind nicht besonders gut für höhere Programmiersprachen geeignet.**
- **Speicherkosten sind häufig in Mikroprozessor-Anwendungen kritisch.**

Das Fehlen höherer Programmiersprachen liegt zum Teil in der Tatsache begründet, daß Mikroprozessoren verhältnismäßig neu sind, und Produkte der Hersteller von Halbleitern sind, anstelle von Computer-Herstellern. Es existieren sehr wenige höhere Sprachen für Mikroprozessoren. Die bekanntesten hiervon sind die Sprachen wie BASIC[4], PASCAL[5], FORTRAN und die PL/I-Sprachen, wie PL/M[6], MPL und PLμS.

Weil nun die wenigen existierenden höheren Sprachen nicht anerkannten Standards entsprechen, kann der Mikroprozessor-Anwender nicht erwarten, daß viele Programme übertragbar sind, er Zugriff zu Programm-Bibliotheken erhält, oder frühere Erfahrungen oder Programme verwenden kann. Die verbleibenden Hauptvorteile liegen daher in der Verringerung des Programmier-Aufwandes und des geringeren erforderlichen detaillierten Verständnisses der Computer-Architektur.

KOSTEN FÜR HÖHERE SPRACHEN

Die mit der Verwendung höherer Programmiersprachen bei Mikroprozessoren verbundenen Kosten sind beträchtlich. Mikroprozessoren sind für Steuerungen und andere Anwendungen besser geeignet als für Zeichen-Manipulationen und Sprach-Analysen bei der Kompilierung. Daher werden die meisten Kompilierer für Mikroprozessoren nicht auf einem System laufen, das auf einem Mikroprozessor basiert. Sie erfordern stattdessen einen wesentlich größeren Computer, das heißt, sie sind eher Cross-Kompilierer (Kompilierer, der nicht auf dem Mikrocomputer läuft, für den das zu kompilierende Programm geschrieben ist) anstatt Selbst-Kompilierer. Ein Anwender muß daher nicht nur die Aufwendungen für den größeren Computer tragen, er muß auch das Programm physikalisch vom großen Computer zum Mikrocomputer übertragen.

Es sind einige wenige Selbst-Kompilierer verfügbar. Diese Kompilierer laufen auf dem Mikrocomputer, für den sie den Objektcode erzeugen. Unglücklicherweise benötigen sie jedoch sehr viel Speicherplatz (16K oder mehr), sowie spezielle unterstützende Hardware und Software.

UNZWECKMÄSSIGKEIT HÖHERER SPRACHEN

Höhere Programmiersprachen sind auch im allgemeinen für Mikroprozessor-Anwendungen nicht gut geeignet. Die meisten der bekannten Sprachen sind entweder zur Lösung wissenschaftlicher Probleme oder für die Handhabung großer Datenmengen in der kommerziellen Datenverarbeitung vorgesehen. Wenige Mikroprozessor-Anwendungen fallen in diese Bereiche. Die meisten Mikroprozessor-Anwendungen beinhalten das Senden von Daten und Steuer-Informationen zu Ausgabe-Bausteinen und das Empfangen von Daten und Status-Informationen von Eingabe-Bausteinen. Häufig bestehen die Steuer- und Status-Informationen aus einigen wenigen binären Ziffern mit sehr genauen Bedeutungen bezüglich der Hardware. Wenn man versuchen würde, ein typisches Steuerprogramm in einer höheren Sprache zu schreiben, würde man sich häufig so fühlen, als ob man versuchen würde, eine Suppe mit Stäbchen zu essen. Für Aufgaben wie etwa Testgeräte, Terminals, Navigationssysteme und Bürogeräte arbeiten die höheren Sprachen wesentlich besser als dies bei Anwendungen in der Instrumentation, Kommunikation, Steuerung von peripheren Bausteine und im Automobil der Fall wäre.

ANWENDUNGSBEREICHE FÜR HÖHERE SPRACHEN

Besser geeignete Anwendungen für höhere Sprachen sind jene, in denen große Speicher erforderlich sind. Wenn die Kosten eines einzelnen Speicherchips wesentlich sind, wie dies zum Beispiel in einem Steuergerät, elektronischen Spiel, Haushaltsgerät oder einem kleineren Instrument der Fall ist, dann ist die Ineffizienz der höheren Sprache nicht mehr tragbar. Wenn andererseits das System mehrere tausend Bytes eines Speichers besitzt, wie dies in einem Terminal oder Testgerät der Fall ist, dann ist die Ineffizienz der höheren Sprachen nicht so wesentlich. Natürlich sind die Größe des Programmes und die produzierte Stückzahl des betreffenden Gerätes ebenfalls wesentliche Faktoren. Ein großes Programm wird die Vorteile der höheren Sprachen entsprechend nutzen. Andererseits werden bei einer Anwendung mit hohen Stückzahlen die festen Software-Entwicklungskosten nicht so wesentlich sein, wie die Speicherkosten, die ein Teil jedes Systems sind.

WELCHES NIVEAU SOLLTE MAN VERWENDEN?

Dies hängt von der speziellen Anwendung ab. Es sollen kurz die Faktoren zusammengefaßt werden, die für ein spezielles Programmier-Niveau den Ausschlag geben:

ANWENDUNGEN FÜR MASCHINENSPRACHE

Maschinensprache

- **Ein Programm in Maschinensprache ist tatsächlich völlig unwirtschaftlich und schwierig zu dokumentieren. Ein Assembler kostet wenig und verringert die Programmierzeit außerordentlich.**

ANWENDUNGEN FÜR ASSEMBLERSPRACHE

Assemblersprache:

- **Kleine bis mittlere Programme**
- **Anwendungen, bei denen die Speicherkosten ein wesentlicher Faktor sind**
- **Echtzeit-Steueranwendungen**
- **Begrenzte Datenverarbeitung**
- **Anwendungen mit hohen Stückzahlen**
- **Anwendungen, in denen mehr Eingaben/Ausgaben oder Steuerungen als Berechnungen vorkommen.**

ANWENDUNGEN FÜR HÖHERE SPRACHEN

Höhere Sprachen

- **Lange Programme**
- **Anwendungen mit kleiner Stückzahl, die jedoch lange Programme benötigen**
- **Anwendungen, die bereits große Speicher erfordern**
- **Anwendungen, in denen mehr Berechnungen als Eingaben/Ausgaben oder Steuerungen vorkommen**
- **Kombatibilität mit ähnlichen Anwendungen, die größere Computer verwenden**
- **Verfügbarkeit von speziellen Programmen in höherer Sprache, die in der vorliegenden Anwendung verwendet werden können**

Viele andere Faktoren sind ebenfalls wichtig, wie etwa die Verfügbarkeit eines größeren Computers für die Entwicklung, Erfahrung mit speziellen Sprachen und Kompatibilität mit anderen Anwendungen.

Wenn schließlich die Hardware die wesentlichsten Kosten in unserer Anwendung darstellt, oder die Geschwindigkeit kritisch ist, sollte auf jeden Fall die Assemblersprache bevorzugt werden. Man muß jedoch damit rechnen, daß man zusätzliche Zeit für die Entwicklung der Software aufwenden muß, um dafür niedrigere Speicherkosten und höherer Ausführungsgeschwindigkeit zu erhalten. Wenn die Software die größten Kosten in unserer Anwendung darstellen, so sollte man höhere Sprachen bevorzugen. Aber es sind zusätzliche Kosten für die Hilfs-Hardware und Software erforderlich.

Natürlich ist auch die Verwendung sowohl der Assemblersprache als auch höherer Programmiersprachen möglich. Man kann das Programm zunächst in einer höheren Sprache schreiben und dann einzelne Abschnitte durch Assemblersprache ersetzten.[7] Es wird dies jedoch in der Praxis nicht häufig durchgeführt, da hierdurch gewaltige Schwierigkeiten bei der Fehlersuche, dem Testen und der Dokumentation entstehen.

WIE SIEHT ES MIT DER ZUKUNFT AUS?

ZUKÜNFTIGE TRENDS IN PROGRAMMIER-SPRACHEN

Es ist zu erwarten, daß in Zukunft eine Tendenz in Richtung höherer Programmiersprachen aus folgenden Gründen vorhanden sein wird:

- Programme scheinen immer aufwendiger und größer zu werden
- Hardware und Speicher werden billiger
- Software und Programmierer gewinnen an Erfahrung,
- Speicherchips bekommen mehr Kapazität bei niedrigeren Kosten "pro Bit", so daß die tatsächlichen Einsparungen an Speicherkosten nicht mehr so stark ins Gewicht fallen.
- Es werden geeignetere und effizientere höhere Programmiersprachen entwickelt.
- Höhere Programmiersprachen werden weiter standardisiert.

Die Programmierung von Mikroprozessoren in Assemblersprache ist jedoch keine aussterbende Kunst, wie es derzeit bei größeren Computern der Fall ist. Jedoch längere Programme, billigere Speicher und teurere Programmierer werden einen immer größeren Anteil der Software-Kosten bei den meisten Applikationen zur Folge haben. Bei zahlreichen Anwendungen werden daher höhere Programmiersprachen Verwendung finden.

WESHALB DIESES BUCH?

Wenn die Zukunft scheinbar den höheren Programmiersprachen gehört, wozu dient dann ein Buch über Programmierung in Assembler-Sprache? Die Gründe sind:

1) Die meisten gegenwärtigen Anwender von Microcomputern programmieren in Assemblersprache (mindestens 2/3 gemäß einer kürzlich erfolgten Umfrage).
2) Viele Anwender von Mikrocomputern werden weiter in Assemblersprache programmieren, da sie deren detaillierte Steuerung benötigen.
3) Bis jetzt ist noch keine allgemein verfügbare oder standardisierte höhere Programmiersprache entwickelt worden.
4) Zahlreiche Anwendungen erfordern die Effizienz der Assemblersprache.
5) Das gründliche Verständnis der Assemblersprache kann bei der Entwicklung höherer Sprachen helfen.

Der Rest dieses Buches wird sich ausschließlich mit Assemblern und Programmierung in Assemblersprache befassen. Es ist jedoch wünschenswert, daß die Leser wissen, daß die Assemblersprache nicht die einzige Alternative ist. Man sollte sorgfältig neue Entwicklungen beobachten, die möglicherweise eine wesentliche Reduzierung der Programmierkosten bewirken, falls derartige Kosten ein wesentlicher Faktor für die jeweilige Anwendung ist.

LITERATUR

1) M. H. Halstead, Elements of Software Science, American Elsevier, New York, 1977.
2) V. Schneider, "Prediction of Software Effort and Project Duration." SIGPLAN Notices, June 1978, pp. 49–55.
3) M. Phister Jr., Data Processing Technology and Economics, Santa Monica Publishing Co., Santa Monica, CA, 1976.
4) Albrecht, Finkel, and Brown, BASIC for Home Computers, Wiley, New York, 1978.
5) K. L. Bowles, Microcomputer Problem Solving Using PASCAL, Springer Verlag, New York, 1977.
6) D. D. McCracken, A Guide to PL/M Programming for Microcomputer Applications, Addison-Wesley, Reading, Mass, 1978.
7) P. Caudill, "Using Assembly Coding to Optimize High-Level Language Programs," Electronics, February 1, 1979, pp. 121–124.

Kapitel 2
ASSEMBLER

In diesem Kapitel werden die von Assemblern ausgeführten Funktionen beschrieben, beginnend mit den Eigenschaften, die den meisten Assemblern gemeinsam sind, bis zu weitergehenden Möglichkeiten wie Makros und bedingte Assemblierung. **Dieses Kapitel kann man jedoch zunächst überfliegen und erst dann ausführlicher lesen, wenn man mit der Materie vertrauter ist.**

EIGENSCHAFTEN VON ASSEMBLERN

Wie bereits eingangs erwähnt, führen moderne Assembler nicht nur die Übersetzung von Mnemoniks in Assemblersprache in die entsprechenden Binärcodes durch. Es soll daher beschrieben werden, wie ein Assembler die Übersetzung von Mnemoniks ausführt, bevor zusätzliche Eigenschaften der Assembler behandelt werden. Abschließend wird erläutert, wie man Assembler verwendet.

ASSEMBLER-BEFEHLE

ASSEMBLER-SPRACHEN-FELDER

Befehle (oder "Anweisungen") der Assemblersprache werden in eine Anzahl von Feldern aufgeteilt, wie aus Tabelle 2-1 zu sehen ist. Das Operationscode-Feld ist das einzige Feld, das niemals leer sein darf. Es enthält immer entweder ein Befehls-Mnemonik oder eine Anweisung für den Assembler, genannt Pseudo-Befehl, Pseudo-Operation oder abgekürzt Pseudo-Op.

Das Operanden- oder Adressenfeld kann eine Adresse oder Daten enthalten, oder auch leer sein.

Die Felder für den Kommentar und das Namensfeld (Labelfield) können nach Bedarf verwendet werden. Der Programmierer kann einen Namen oder eine Markierung (label) einem Befehl zuweisen oder einen Kommentar zur persönlichen Bequemlichkeit verwenden, z.B. um das Programm leichter verwendbar und lesbar zu machen.

Tabelle 2-1. Die Felder eines Assemblersprachen-Befehls.

Marken-Feld	Operationscode oder Mnemonik-Feld	Operanden- oder Adressen-Feld	Kommentar-Feld
START	LDA	VAL1	;LADE ERSTE ZAHL IN A
	ADC	VAL2	;ADDIERE ZWEITE ZAHL ZU A
	STA	SUM	;SPEICHERE SUMME
NEXT	?	?	;NÄCHSTER BEFEHL
.			
.			
.			
VAL1	*=*+1		
VAL2	*=*+1		
SUM	*=*+1		

Tabelle 2-2. Standard-Begrenzungszeichen des 6502-Assemblers.

'Zwischenraum'	zwischen Markierung (Label) und Operationscode und zwischen Operationscode und Adresse
,	zwischen Operanden im Adressen-Feld
; oder !	vor einem Kommentar

Beachten Sie, daß 6502-Assembler sehr stark variieren und einige diese Begrenzer nicht verwenden.

FORMAT

Natürlich muß dem Assembler auf irgendeine Weise mitgeteilt werden, wo ein Feld endet und das nächste beginnt.
Assembler, die mit einer Eingabe über Lochkarten arbeiten, fordern häufig , daß jedes Feld in einer speziellen Kartenspalte beginnt. Dies stellt ein **festes Format** dar. Feste Formate sind jedoch unbequem und lästig für den Programmierer. Die Alternative hierzu ist ein **freies Format**, bei dem die Felder an beliebiger Stelle auf einer Zeile erscheinen können.

BEGRENZUNGSZEICHEN

Wenn der Assembler die Position auf der Zeile zur Trennung der Felder nicht verwenden kann, so muß er etwas anderes zu Hilfe nehmen.
Die meisten Assembler benützen ein spezielles Symbol oder Begrenzungs-Zeichen (delimiter) am Beginn oder Ende jedes Feldes. Das am meisten gebräuchliche Begrenzungszeichen ist das Zeichen für den Zwischenraum. Kommata, Punkte, Strichpunkte, Schrägstriche, Fragezeichen und andere Zeichen, die anderweitig nicht verwendet werden, können in Programmen in Assemblersprache eingesetzt werden und als Begrenzungszeichen dienen. Tabelle 2-2 enthält die des 6502 standardisierten Begrenzungszeichen.

Mit Begrenzungszeichen muß man jedoch etwas vorsichtig sein. Manche Assembler sind sehr empfindlich gegen zusätzliche Zwischenräume oder das Auftreten von Begrenzungszeichen in Kommentaren oder Marken. Ein gut geschriebener Assembler wird mit diesen kleineren Problemen leicht fertig, aber manche Assembler sind nicht so gut abgefaßt. Um derartige Probleme zu vermeiden, können folgende Regeln behilflich sein:

1) Verwenden Sie keine zusätzlichen Zwischenräume, speziell nach Kommata, mit denen Operanden getrennt werden.
2) Verwenden Sie keine Begrenzungszeichen in Namen oder Marken.
3) Verwenden Sie Standard-Begrenzungszeichen auch dann, wenn Ihr Assembler diese nicht benötigt. Dann werden diese Programme für jeden Assembler annehmbar.

MARKEN (LABELS)

MARKEN-FELD

Das Markenfeld ist das erste Feld eines Befehls in Assemblersprache. Es kann auch leer sein.
Wenn eine Marke vorliegt, dann definiert der Assembler die Marke als äquivalent mit der Adresse des Speicherplatzes, in den das erste Byte des Objekt-Programmes, das sich aus diesem Befehl ergibt, geladen wird. Man kann danach die Marke als Adresse oder als Daten in einem anderen Adressenfeld eines Befehls verwenden. Der Assembler wird die Marke durch den zugewiesenen Wert ersetzen, wenn er ein Objektprogramm erzeugt.

MARKEN IN SPRUNG-BEFEHLEN

Marken werden am häufigsten in Sprung- Aufruf- oder Verzweigungs-Befehlen verwendet. Diese Befehle bringen einen neuen Wert in den Befehlszähler und ändern somit die normale sequentielle Ausführung von Befehlen.
JUMP 150_{16} bedeutet "Plaziere den Wert 150_{16} in den Befehlszähler". Der nächste auszuführende Befehl wird derjenige im Speicherplatz 150_{16} sein. Der Befehl JUMP START bedeutet "Plaziere den Wert, der der Marke START zugewiesen wurde, in den Befehlszähler". Der nächste auszuführende Befehl wird derjenige im Speicherplatz sein, der zur Marke START gehört. Tabelle 2-3 enthält ein Beispiel.

Tabelle 2-3. Zuweisung und Verwendung einer Markierung.

ASSEMBLERSPRACHEN-PROGRAMM	
START	LADE AKKUMULATOR MIT 100
	.
	.
	.
	. (HAUPT-PROGRAMM)
	.
	.
	.
	SPRINGE ZU START

Wenn die Maschinensprachen-Version dieses Programms ausgeführt wird, so bewirkt der Befehl SPRINGE ZU START, daß die Adresse des mit START markierten Befehls in den Befehlszähler plaziert wird. Dieser Befehl wird dann ausgeführt.

Weshalb werden Marken verwendet? Hier sind einige Gründe:

1) Eine Marke erleichtert das Auffinden und Erinnern einer Programmstelle.
2) Die Marke kann für die Korrektur eines Programmes verschoben werden. Man braucht einen darauf folgenden Befehl, der die Marke verwendet, nicht ändern. Der Assembler führt alle erforderlichen Änderungen automatisch selbst aus.
3) Der Assembler oder Lader kann das ganze Programm verschieben, indem er eine Konstante (eine sogenannte Verschiebungs-Konstante) zu jeder Adresse addiert, der eine Marke zugewiesen wurde. Man kann daher das Programm verschieben, um das Einsetzen anderer Programme zu gestatten oder um einfach den Speicher neu zu ordnen.

VERSCHIE-BUNGS-KONSTANTE

4) Das Programm ist einfacher in der Anwendung als ein Bibliotheks-Programm, d.h. es kann jemand unser Programm einfacher übernehmen und es zu einem völlig anderen Programm hinzufügen.
5) Man braucht sich keine Speicher-Adressen auszudenken. Die Festlegung von Speicheradressen ist besonders schwierig mit Mikroprozessoren, die Befehle mit unterschiedlicher Länge besitzen.

Sie sollten Jedem Befehl eine Marke zuweisen , auf den Sie sich vielleicht später beziehen wollen.

AUSWAHL VON MAKREN

Die nächste Frage ist, welche Marken man verwenden soll. Durch die Assembler-Syntax wird häufig die Anzahl der Zeichen begrenzt (gewöhnlich 5 oder 6), das erste Zeichen muß meist ein Buchstabe sein, und die folgenden Zeichen müssen meist Buchstaben, Zahlen oder ein spezielles Zeichen sein. Innerhalb dieser Begrenzungen hat man freie Wahl.

Man verwendet vorteilhafterweise Markierungen, die auf ihre Verwendung schließen lassen, d.h. mnemotechnische Marken. Typische Beispiele hierfür wären ADDW in einem Programm, bei dem ein Wort zu einer Summe addiert wird, SRETX in einem Programm, in dem nach dem ASXCII-Zeichen ETX gesucht (search for) wird, oder NKEYS für einen Platz im Datenspeicher, das die Anzahl der Tasteneingaben (number of key entries) enthält. Wenn sich aus der Bezeichnung einer Marke auf ihre Verwendung schließen läßt, so kann man sich diese leichter merken, und sie vereinfachen die Programm-Dokumentation. Manche Programmierer ziehen es vor, ein Standard-Format für Marken zu verwenden, die beispielsweise mit L0000 beginnen. Dadurch ergibt sich bei diesen Marken von selbst die entsprechende Reihenfolge (wobei man einige Zahlen überspringen kann, um nachträgliche Einfügungen zu ermöglichen); sie machen jedoch die Dokumentation nicht sehreinfach.

REGELN FÜR MARKIE-RUNGEN

Einige einfache Regeln für die Auswahl von Marken vermeiden Schwierigkeiten. Die Einhaltung folgender Regeln ist empfehlenswert:

1) Verwenden Sie keine Marken, die mit Operationscodes oder anderen Mnemoniks übereinstimmen. Die meisten Assembler gestatten eine derartige Verwendung nicht. Bei anderen ist dies zwar der Fall, aber es kann zu Verwechslungen führen.
2) Verwenden Sie keine Marken die länger sind, als die vom Assembler zugelassen. Assembler besitzen verschiedene Kürzungsregeln.
3) Vermeiden Sie spezielle Zeichen (nicht-alphabetische und nicht-numerische), sowie Kleinbuchstaben. Manche Assembler gestatten deren Verwendung nicht, andere verwenden einige hiervon. Man beschränkt sich am besten auf Großbuchstaben und Zahlen.
4) Beginnen Sie jede Markierung mit einem Buchstaben. Derartige Marken werden immer angenommen.
5) Verwenden Sie keine Marken, die sich mit anderen verwechseln lassen. Vermeiden Sie die Buchstaben I, O, Z und die Zahlen 0, 1 und 2. Vermeiden Sie ferner Anordnungen wie XXXX und XXXXX, sie lassen sich schwer unterscheiden.
6) Wenn Sie nicht sicher sind, daß eine Marke gestattet ist, dann verwenden Sie diese auch nicht. Es ist kein besonderer Vorteil genau zu erfahren, was der Assembler annehmen wird.

Dies sind gutgemeinte Empfehlungen. Man muß sie nicht befolgen, kann aber auf diese Weise unangenehme Probleme vermeiden.

ASSEMBLER-OPERATIONSCODES (MNEMONIKS)

Die hauptsächliche Aufgabe des Assemblers ist die Übersetzung von Mnemoniks in ihre äquivalenten binären Operationscodes. Der Assembler führt dies unter Verwendung einer festgelegten Tabelle aus, ebenso wie man es bei einer händischen Assemblierung tun würde.

Der Assembler muß jedoch mehr leisten, als nur die Übersetzung der Operationscodes. Er muß irgendwie bestimmen, wieviele Operanden der Befehl benötigt und welcher Art diese sind. Dies kann ziemlich komplex sein, da manche Befehle (wie ein HALT) keine Operanden besitzen. Andere dagegen (wie etwa ein Additions- oder Sprungbefehl) besitzen einen Operanden, während wiederum andere (wie ein Transfer zwischen Registern oder eine Mehr-Byte-Verschiebung) dagegen zwei erfordern. Manche Befehle gestatten sogar alternative Anwendungen, z.B. besitzen manche Computer Befehle (wie Verschieben oder Löschen), die sich entweder auf den Akkumulator oder auf einen Speicherplatz beziehen. Es soll nicht weiter besprochen werden, wie ein Assembler diese Unterscheidungen trifft, wir wollen nur festhalten daß er dies tun muß.

PSEUDO-OPERATIONEN

PSEUDO-OPERATIONEN

Manche Assemblersprachen-Befehle werden nicht direkt in Maschinensprachen-Befehle übersetzt. Diese Befehle stellen "Anweisungen" für den Assembler dar.
Sie weisen das Programm bestimmten Speicherbereichen zu, definieren Symbole, bestimmten Bereichen im RAM für zeitweilige Datenspeicherung, plazieren Tabellen oder andere feste Daten in den Speicher und führen andere Haushaltungsfunktionen aus.

Um diese Anweisungen oder Pseudo-Operationen zu verwenden, plaziert ein Programmierer die Mnemoniks der Pseudo-Operationen in das Operationscode-Feld und eine Adresse oder Daten in das Adressenfeld, wenn eine bestimmte Pseudo-Operation dies erfordert.

Die am meisten gebräuchlichsten Pseudo-Operationen sind:

DATA
EQUATE (=) oder DEFINE
ORIGIN
RESERVE

Verbindende Pseudo-Operationen (zur Verbindung getrennter Programme) sind:

ENTRY
EXTERNAL

Verschiedene Assembler verwenden auch unterschiedliche Bezeichnungen für diese Operationen, ihre Funktionen sind jedoch die gleichen. Pseudo-Operationen für Haushaltungsfunktionen sind folgende:

END
LIST
NAME
PAGE
SPACE
TITLE

Wir werden diese Pseudo-Operationen kurz besprechen, obwohl ihre Funktionen gewöhnlich offensichtlich sind.

DIE PSEUDO-OPERATION DATA

Die Pseudo-Operation DATA gestattet dem Programmierer die Eingabe fester Daten in den Programmspeicher. Diese Daten können enthalten:

Nachschlag-Tabellen
Code-Umwandlungstabellen
Nachrichten
Synchronisationsmuster
Schwellwerte
Namen
Koeffizienten für Gleichungen
Kommandos
Umrechnungs-Faktoren
Bewertungs-Faktoren
Charakteristische Zeiten oder Frequenzen
Unterprogramm-Adressen
Schlüssel-Identifizierungen
Testmuster
Muster für Zeichenerzeugung
Identifizierungsmuster
Gebührentabellen
Standardformen
Maskier-Muster
Zustands-Übergangstabellen

Die Pseudo-Operation DATA behandelt die Daten als ständigen Teil des Programmes.

Das Format der Pseudo-Operation DATA ist gewöhnlich sehr einfach. Ein Befehl wie

```
DZCON    DATA    12
```

wird die Zahl 12 in den nächsten verfügbaren Speicherplatz bringen und diesem Platz die Bezeichnung DZCON zuweisen. Gewöhnlich besitzt jede Data-Pseudo-Operation eine Marke, es sei denn, sie ist eine aus einer Serie von Data-Pseudo-Operationen. Die Daten und Marken können in jeder beliebigen Form vorliegen, die vom Assembler zugelassen ist.

Die meisten Assembler gestatten umfangreichere DATA-Befehle, mit denen eine große Anzahl von Daten zur gleichen Zeit gehandhabt werden können, z.B.:

```
EMESS   DATA   'ERROR'
SQRS    DATA   1,4,9,16,25
```

Ein einzelner Befehl kann zahlreiche Worte des Programmspeichers füllen und wird nur durch die Länge einer Zeile begrenzt. Wenn man nicht alle auf einer Zeile unterbringt, kann immer ein DATA-Befehl auf den anderen folgen, z.B.:

```
MESSG DATA    'NOW IS THE'
      DATA    'TIME FOR ALL'
      DATA    'GOOD MEN'
      DATA    'TO COME TO THE'
      DATA    'AID OF THEIR'
      DATA    'COUNTRY'
```

Assembler für Mikroprozessoren besitzen normalerweise einige Varianten der Standard-Pseudo-Operaton DATA. DEFINE BYTE oder FORM CONSTANT BYTE handhaben 8-Bit-Zahlen. DEFINE WORD oder FORM CONSTANT WORD handhaben 16-Bit-Zahlen oder Adressen. Andere spezielle Pseudo-Operationen können zeichencodierte Daten verarbeiten.

DIE PSEUDO-OPERATION EQUATE (oder DEFINE)

DEFINITION VON NAMEN

Die Pseudo-Operation EQUATE gestattet dem Programmierer das "Gleichsetzen" von Namen und Adressen oder Daten. Diese Pseudo-Operation wird meist mit dem Mnemonik EQU oder = bezeichnet.
Die Namen können sich auf Baustein-Adressen, numerische Daten, Start-Adressen, feste Adressen etc. beziehen.

Die Pseudo-Operation EQUATE weist den numerischen Wert in ihrem Operandenfeld dem Namen in ihrem Markenfeld zu. Hier sind zwei Beispiele:

```
TTY     EQU     5
LAST    EQU     5000
```

Die meisten Assembler gestatten die Definition einer Marke mit den Ausdrücken einer anderen, z.B.:

```
LAST    EQU     FINAL
ST1     EQU     START+1
```

Die Marke im Operandenfeld muß natürlich vorher definiert werden. Häufig kann das Operandenfeld komplexere Ausdrücke enthalten, wie wir später sehen werden. Die Zuweisung doppelter Namen (zwei Namen für die gleichen Daten oder Adressen) kann sehr nützlich sein, wenn man Programme zusammenlegen will, die verschiedene Namen für die gleiche Variable (oder unterschiedliche Ausdrucksweise für scheinbar gleiche Namen) verwenden.

SYMBOL-TABELLE

Es ist zu beachten, daß eine EQU-Pseudo-Operation nicht bewirkt, daß der Assembler irgend etwas in den Programmspeicher plaziert. Der Assembler plaziert einfach einen zusätzlichen Namen in eine Tabelle (eine sogenannte Symbol-Tabelle). Diese Tabelle, anders als die Mnemoniks-Tabelle, muß in einem RAM liegen, da sie sich mit jedem Programm ändert. Das Assembler-Programm wird immer einen Teil eines RAMs benötigen, um die Symbol-Tabelle aufzubewahren. Je mehr RAM zur Verfügung steht, desto mehr Symbole kann sie aufnehmen. Dieses RAM wird zusätzlich zu allem benötigt, was der Assembler an zeitweiligem Speicher braucht.

VERWENDUNG VON NAMEN

Wann verwendet man einen Namen? Die Antwort ist, wann immer man einen Parameter hat, den man ändern möchte oder der irgend eine Bedeutung neben seinem gewöhnlichen numerischen Wert besitzt. Gewöhnlich erfolgt eine Zuweisung von Namen zu Zeit-Konstanten, Baustein-Adressen, Maskier-Mustern, Umwandlungsfaktoren und ähnlichen. Ein Name wie DELAY, TTY, KBD, KROW oder OPEN macht es nicht nur einfacher, den Parameter zu ändern, sondern erleichtert auch die Programm-Dokumentation. Man weist Namen auch Speicherplätzen zu, die einen speziellen Zweck haben. Sie können Daten aufbewahren, den Start des Programms markieren, oder für unmittelbare Speicherung zur Verfügung stehen.

AUSWAHL VON NAMEN

Welche Namen verwendet man? Man verwendet am besten Namen nach den gleichen Grundsätzen wie bei Markierungen, wobei hier besonders bedeutungsvolle Namen empfehlenswert sind. Weshalb soll man den Fernschreiber (teletypewriter) nicht TTY anstatt X15 nennen, eine Bit-Zeitverzögerung BTIME oder BTDLY anstatt WW, die Nummer der "GO"-Taste (key) auf einer Tastatur GOKEY anstatt "PFERD?" Diese Ratschläge scheinen trivial zu sein, jedoch eine überraschende Anzahl von Programmierern verwenden sie nicht.

PLAZIERUNG VON DEFINITIONEN

Wohin wird man die Equate-Pseudo-Operationen plazieren? Die beste Stelle ist am Beginn des Programms, unter entsprechenden Kommentar-Überschriften wie E/A-Adressen, zeitweilige Speicherung, Zeitkonstanten oder Programmstellen. Dadurch lassen sich Definitionen leichter finden, wenn man sie ändern möchte. Ferner wird ein anderer Anwender imstande sein, alle Definitionen an einer zentralen Stelle zu übersehen. Offensichtlich verbessert diese Praxis auch die Dokumentation und erleichtert die Verwendung des Programmes.

Definitionen, die nur in einem speziellen Unterprogramm verwendet werden, sollten am Beginn des Programmes aufscheinen.

DIE PSEUDO-OPERATION ORIGIN

Die ORIGIN Pseudo-Operation (meist abgekürzt ORG) gestattet dem Programmierer das Assemblieren von Programmen, Unterprogrammen oder Daten überall im Speicher. Programme und Daten können in verschiedenen Speicherbereichen liegen, abhängig von der Speicher-Konfiguration. Start-Routinen, Unterbrechungs-Serviceroutinen und andere notwendige Programme können über den ganzen Speicher verteilt sein oder an bestimmten Adressen festliegen.

STELLEN-ZÄHLER

Der Assembler enthält einen Stellenzähler (Location Counter), der vergleichbar ist mit dem Befehlszähler des Computers, und der die Speicherstelle des nächsten zu verarbeitenden Befehls oder Daten enthält. Eine ORG-Pseudo-Operation bewirkt, daß der Assembler einen neuen Wert in den Stellenzähler plaziert, gerade so wie ein Sprungbefehl bewirkt, daß die CPU einen neuen Wert in den Befehlszähler bringt. Die Ausgabe des Assemblers braucht nicht nur Befehle und Daten zu enthalten, sondern muß auch dem Ladeprogramm angeben, wo es die Befehle und Daten in den Speicher plazieren soll.

Mikroprozessor-Programme enthalten häufig verschiedene ORIGIN-Anweisungen für folgende Zwecke:

Rücksetz (Start)-Adressen
Unterbrechungs-Service-Adressen
Adressen von Fallen
RAM-Speicher
Speicherstapel
Haupt-Programm
Unterprogramme
Speicheradressen, reserviert für Eingabe/Ausgabe-Bausteine oder spezielle Funktionen.

Weitere ORIGIN-Anweisungen können Platz für spätere Einfügungen schaffen, Tabellen oder Daten in den Speicher plazieren, oder freien RAM-Speicherraum Datenpuffern zuweisen. Programm- und Datenspeicher in Mikrocomputern können weit verstreute Adressen belegen, um die Hardware-Entwicklung zu vereinfachen.

Typische ORIGIN-Anweisungen sind:

```
ORG     RESET
ORG     1000
ORG     INT3
```

Manche Assembler nehmen standardmäßig einen Wert von Null als ORIGIN, wenn der Programmierer keine ORG-Anweisung an den Beginn des Programmes legt. Das ist bequem, wir empfehlen jedoch die Verwendung einer ORG-Anweisung um Verwirrung zu vermeiden.

DIE PSEUDO-OPERATION RESERVE

RAM-ZUWEISUNG

Die Reserve-Pseudo-Operation gestattet dem Programmierer die Zuweisung von RAM-Bereichen für verschiedene Zwecke wie Daten-Tabellen, zeitweilige Speicherung, indirekte Adressen, einen Stapel etc.

Bei der Verwendung der Pseudo-Operation RESERVE weist man einen Namen einem Speicherbereich zu und gibt die Anzahl der zuzuweisenden Speicherplätze an. Hier sind einige Beispiele:

```
NOKEY   RESERVE  1
TEMP    RESERVE  50
VOLTG   RESERVE  80
BUFR    RESERVE  100
```

Man kann die Pseudo-Operation RESERVE zur Reservierung von Speicherplätzen im Programmspeicher oder im Datenspeicher verwenden. Die RESERVE Pseudo-Operation besitzt jedoch eine größere Bedeutung, wenn sie auf den Datenspeicher angewendet wird.

In der Praxis erhöhen alle RESERVE Pseudo-Operationen nicht den Stellenzähler (Location Counter) des Assemblers durch den Betrag, der im Operandenfeld angegeben ist. Der Assembler erzeugt tatsächlich überhaupt keinerlei Objektcode.

Beachten Sie die folgenden Eigenschaften von RESERVE:

1) Die Markierung der RESERVE-Pseudo Operation wird dem Wert der ersten reservierten Adresse zugewiesen. Beispielsweise reserviert die Sequenz:

   ```
   TEMP    RESERVE    20
   ```

   20 Bytes des RAMs und weist dem Namen TEMP der Adresse des ersten Bytes zu.
2) Man muß die Anzahl der zu reservierenden Speicherstellen spezifizieren. Es gibt hier keinen Standardfall.
3) Es werden keine Daten in die reservierten Speicherstellen plaziert. Wenn sich Daten zufällig in diesen Speicherstellen befinden, werden sie dort belassen.

INITIALISIERUNG DES RAMs

Einige Assembler gestatten dem Programmierer die Plazierung von Anfangswerten in das RAM. Es ist dringend zu empfehlen, daß man diese Eigenschaft nicht verwendet. Es wird hierbei angenommen, daß das Programm (zusammen mit den Anfangswerten) von einem externen Gerät geladen wird (z.B. Lochstreifen oder Floppy Disks), jedesmal, wenn es abläuft. Die meisten Mikroprozessor-Programme befinden sich andererseits in nicht-flüchtigen ROMs und starten beim Einschalten der Betriebsspannung. In derartigen Situationen behält das RAM seinen Inhalt nicht, und es wird auch nicht neu geladen. Daher sind immer Befehle vorzusehen, die das RAM in dem eintsprechenden Programm initialisieren.

BINDE-PSEUDO-OPERATIONEN

EXTERNE REFERENZEN

Wir brauchen häufig Anweisungen in einem Programm oder Unterprogramm, die Namen verwenden, die anderswo definiert wurden. Derartige Namen nennt man externe Referenzen.
Hierzu ist ein spezielles Binde-Programm (linking program) erforderlich, um die Werte tatsächlich einzusetzen und zu bestimmen, ob irgendwelche Namen nicht oder doppelt definiert wurden.

Die Pseudo-Operation EXTERNAL, gewöhnlich EXT abgekürzt, zeigt an, daß der Name anderswo definiert wurde.

Die Pseudo-Operation ENTRY, gewöhnlich ENT abgekürzt, zeigt an, daß der Name für den Gebrauch an beliebiger Stelle verfügbar ist, d.h. er ist in diesem Programm definiert.

Die genaue Art und Weise, in der Binde-Pseudo-Operationen ausgeführt werden, variiert sehr weit von Assembler zu Assembler. Wir werden uns daher auf derartige **Pseudo-Operationen** weiterhin nicht beziehen, sie **sind jedoch in praktischen Anwendungen sehr nützlich.**

HAUSHALTUNGS-PSEUDO-OPERATIONEN

Es gibt verschiedene Haushaltungs-Pseudo-Operationen, die den Betrieb des Assemblers und seine Programm-Auflistung eher beeinflussen, als das Ausgangsprogramm selbst. Gebräuchliche Haushaltungs-Pseudo-Opertationen beinhalten:

- END, durch die das Ende des Quellenprogrammes in Assemblersprache markiert wird.
- LIST, das dem Assembler sagt, das Quellenprogramm zu drucken. Manche Assembler gestatten Variationen wie NO LIST oder LIST SYMBOL.
- NAME oder TITLE, womit ein Name oben auf jede Seite der Auflistung gedruckt wird.
- PAGE oder SPACE, wodurch zur nächsten Seite oder nächsten Zeile gesprungen wird, und das Aussehen der Auflistung verbessert und sie dadurch lesbarer macht.
- PUNCH, wodurch der aufeinanderfolgende Objektcode auf den Lochstreifen-Stanzer übertragen wird. Die Pseudo-Operation kann in vielen Fällen eine Standard-Option sein, und ist deshalb nicht notwendig.

MARKIERUNGEN BEI PSEUDO-OPERATIONEN

Anwender möchten häufig gerne wissen, ob oder wann sie eine Markierung einer Pseudo-Operation zuweisen können. Hierzu ist folgendes zu empfehlen:

1) Alle EQUATE-Pseudo-Operationen müssen Markierungen besitzen. Sie haben andernfalls keinen Sinn, da ihr Zweck in der Definition der Bedeutung dieser Markierung besteht.
2) DATA- und RESERVE-Pseudo-Operationen sollten gewöhnlich Markierungen besitzen. Die Markierung identifiziert den ersten verwendeten oder zugewiesenen Speicherplatz.
3) Andere Pseudo-Operationen sollten keine Markierungen haben. Manche Assembler gestatten, daß derartige Pseudo-Operationen Markierungen besitzen, die Bedeutung der Markierungen ist jedoch unterschiedlich und nicht standardisiert. Diese Praxis ist nicht zu empfehlen.

ADRESSEN UND DAS OPERANDENFELD

Die meisten Assembler gestatten dem Programmierer ziemlich viel Freiheit bei der Beschreibung des Inhalts des Operanden- oder Adressenfeldes. Aber erinnern wir uns daran, daß der Assembler eigene Namen für Register und Befehle besitzt und andere eigene Namen besitzen kann.

Einige gebräuchliche Optionen für das Operandenfeld sind:

DEZIMAL-DATEN ODER ADRESSEN

1) Dezimalzahlen

Die meisten Assembler nehmen an, daß alle Zahlen Dezimalzahlen sind, außer sie sind anderweitig markiert. Daher bedeutet:

```
ADD     100
```

"Addiere den Inhalt des Speicherplatzes 100_{10} zum Inhalt des Akkumulators".

NICHT-DEZIMALE ZAHLEN-SYSTEME

2) Andere Zahlensysteme

Die meisten Assembler nehmen auch binäre, oktale oder hexadezimale Eingaben an. Man muß jedoch derartige Zahlensysteme auf irgendeine Weise identifizieren, z.B. indem man der Zahl ein Identifikationszeichen oder Buchstaben vorausgehen läßt. Hier sind einige gebräuchliche Kennzeichnungen:

B oder % für binär
O,@, Q, oder C für oktal (gewöhnlich vermeidet man jedoch den Buchstaben O, damit keine Verwechslung mit 0 entsteht).
H oder $ für hexadezimal (oder Standard-BCD)
D für dezimal. D kann jedoch weggelassen werden, da es ein Standardfall ist.

Assembler benötigen im allgemeinen Hexadezimalzahlen, um mit einer Ziffer zu beginnen (z.B. 0A36 anstatt A36), damit sie zwischen Zahlen und Namen oder Markierungen unterscheiden können. Es ist empfehlenswert, Zahlen in der Basis einzugeben, in der ihre Bedeutung am deutlichsten ist, d.h. dezimale Konstanten in dezimal, Adressen und BCD-Zahlen in hexadezimal, Maskier-Muster oder Bit-Ausgänge in binär, wenn sie kurz sind und in hexadezimal, wenn sie lang sind.

3) Namen

Namen können im Operandenfeld erscheinen. Sie werden wie Daten behandelt, die sie darstellen. **Aber erinnern wir uns daran, daß es einen Unterschied zwischen Daten und Adressen gibt.** Die Sequenz

```
FIVE  EQU   5
      ADDA  FIVE
```

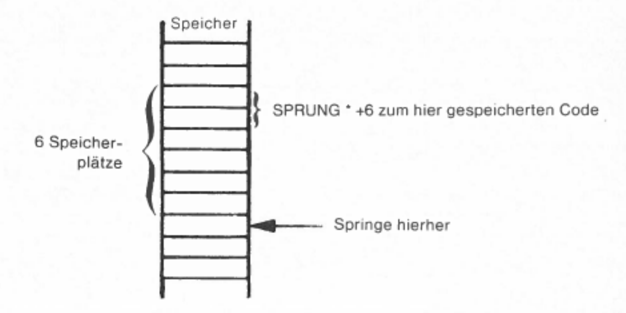
wird den Inhalt des Speicherplatzes 0005 (nicht notwendigerweise die Zahl 5) zum Inhalt des Akkumulators A addieren.

4) Der momentane Wert des Speicherzählers (gewöhnlich bezeichnet als ·oder $).

Dies ist vor allem bei Sprungbefehlen von Bedeutung. Z.B. bewirkt

JUMP ·+6

einen Sprung zum Speicherplatz sechs Worte hinter dem Wort, das das erste Byte des Sprungbefehles enthält:



Die meisten Mikroprozessoren besitzen Zwei- und Dreiwort-Befehle. Daher wird man Schwierigkeiten bei der genauen Bestimmung haben, wie weit voneinander entfernt zwei Assemblersprachen-Anweisungen sind. Die Verwendung von Versetzungen (offsets) vom Stellenzähler resultieren daher häufig in Fehlern, die man vermeiden kann, wenn man stattdessen Markierungen verwendet.

ASCII-ZEICHEN

5) Zeichencodes
Die meisten Assembler gestatten die Eingabe von Text als ASCII-Zeichenfolgen (strings). Derartige Zeichenfolgen können sich entweder zwischen einfachen oder doppelten Anführungszeichen befinden. Zeichenfolgen können auch ein Anfangs- oder Abschlußsymbol verwenden, wie etwa A oder C. Einige wenige Assembler gestatten auch EBCDIC-Zeichenfolgen.
Es ist zu empfehlen, daß man Zeichenfolgen für alle Arten von Text verwendet. Es verbessert die Deutlichkeit und Lesbarkeit des Programms.

ARITHMETISCHE UND LOGISCHE AUSDRÜCKE

6) Kombination von 1 bis 5 mit arithmetischen, logischen oder speziellen Operatoren. Nahezu alle Assembler gestatten einfache arithmetische Kombinationen, wie etwa START+1. Einige Assembler erlauben auch eine Multiplikation, Division, logische Funktionen, Verschiebungen, etc. Diese werden als Ausdrücke (expressions) bezeichnet. Es ist zu beachten, daß der Assembler Ausdrücke während der Assemblierzeit prüft. Auch wenn ein Ausdruck im Operandenfeld Multiplikationen enthalten kann, so wird man nicht imstande sein, die Multiplikation in der Logik des eigenen Programmes zu verwenden, außer man schreibt ein Unterprogramm für diesen speziellen Zweck.

Assembler unterscheiden sich darin, welche Ausdrücke sie annehmen und wie sie diese interpretieren. Komplexe Ausdrücke ergeben schwer lesbare und schwer verständliche Programme.

Während dieses Abschnittes wurden mehrere Empfehlungen gegeben, sie sollen hier wiederholt und ergänzt werden. **Im Allgemeinen soll man auf Deutlichkeit und Einfachheit achten.** Es lohnt sich nicht, ein Experte für besondere Feinheiten des Assemblers zu sein, oder sehr komplizierte Ausdrücke zu verwenden, wenn dies nicht erforderlich ist.
Es ist folgendes zu empfehlen:

- Verwenden Sie das deutlichste Zahlensystem oder Zeichencode für Daten.
- Masken und BCD-Zahlen in dezimal, ASCII-Zeichen in oktal, oder gewöhnliche numerische Konstanten in hexadezimal haben keinen Zweck und sollten daher nicht verwendet werden.
- Vergessen Sie nicht, Daten und Adressen zu unterscheiden.
- Verwenden Sie keine Versetzungen (offsets) vom Stellenzähler.
- Halten Sie Ausdrücke einfach und deutlich. Verwenden Sie keine ausgefallenen Eigenschaften des Assemblers.

BEDINGTE ASSEMBLIERUNG

Manche Assembler gestatten das Einschließen oder Ausschließen von Teilen des Quellprogramms, abhängig von Bedingungen, die zur Assemblierzeit vorhanden sind. Dies wird bedingte Assemblierung genannt. Sie gibt dem Assembler etwas von der Flexibilität eines Kompilierers. **Die meisten Mikrocomputer-Assembler haben begrenzte Möglichkeiten für eine bedingte Assemblierung.**

```
IF COND
.
.(CONDITIONAL PROGRAM)
.
.
ENDIF
```

Wenn der Ausdruck COND während der Assemblierzeit gültig ist, dann sind die Befehle zwischen IF und ENDIF (zwei Pseudo-Operationen) im Programm enthalten.

Typische Anwendungen von bedingter Assemblierung sind:

1) Das Einschließen oder Weglassen zusätzlicher Variablen.
2) Das Unterbringen von Diagnostik-Routinen in Testläufen.
3) Die Verwendung von Daten mit unterschiedlichen Bit-Längen.
4) Die Schaffung spezieller Versionen eines allgemeinen Programmes.

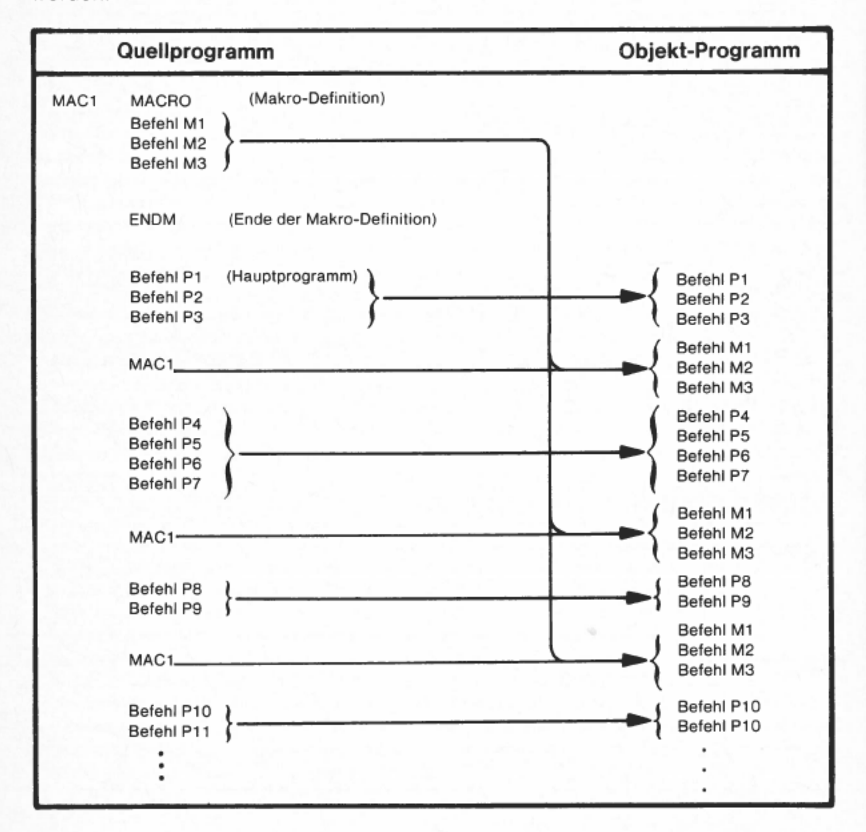
Unglücklicherweise tendiert die bedingte Assemblierung zum Verwirren von Programmen und machten sie schwierig zu lesen. Es soll daher die bedingte Assemblierung nur verwendet werden, falls dies erforderlich ist.

MAKROS

DEFINITION EINER SEQUENZ VON BEFEHLEN

Man wird häufig finden, daß spezielle Sequenzen von Befehlen mehrfach in einem Quellprogramm aufscheinen. Wiederholte Befehls-Sequenzen können Forderungen unserer Programm-Logik wiedergeben, oder sie können Unvollkommenheiten in dem Befehlssatz unseres Mikroprozessors ausgleichen. Man kann das wiederholte Schreiben der gleichen Befehls-Sequenz vermeiden, wenn man ein Makro verwendet.

Makros gestatten die Zuweisung eines Namens zu einer Befehls-Sequenz. Man kann dann den Makro-Namen im Quellprogramm verwenden, anstatt der wiederholten Befehls-Sequenz. Der Assembler wird den Makro-Namen durch die entsprechende Sequenz von Befehlen ersetzen. Dies kann wie folgt dargestellt werden:



Makros sind nicht dasselbe wie Unterprogramme. Ein Unterprogramm tritt nur einmal in einem Programm auf, und die Programm-Ausführung verzweigt zu diesem Unterprogramm. Ein Makro wird zu einer tatsächlichen Befehls-Sequenz erweitert, jedes Mal wenn der Makro auftritt. Daher bewirkt der Makro keine Verzweigung.

VORTEILE VON MAKROS

Makros besitzen folgende Vorteile:

1) Kürzere Quellprogramme
2) Bessere Programm-Dokumentation.
3) Verwendung von fehlerfreien Befehls-Sequenzen. Sobald der Makro fehlerfrei gemacht wurde, ist man sicher, daß eine einwandfreie Befehls-Sequenz bei jeder Verwendung des Makros zur Verfügung hat.
4) Einfacher Austausch. Man ändert die Makro-Definition und der Assembler führt die Änderung aus, jedesmal wenn der Makro verwendet wird.
5) Einschließen von Kommandos, Schlüsselworten oder anderen Computer-Befehlen in den grundlegenden Befehlssatz. Man verwendet Makros zur Erweiterung oder Erläuterung des Befehlssatzes.

NACHTEILE VON MAKROS

Die Nachteile von Makros sind:

1) Wiederholung der gleichen Befehls-Sequenzen.
2) Ein einzelner Makro kann eine Vielzahl von Befehlen bilden.
3) Mangel an Standardisierung.
4) Mögliche Einflüsse auf Register und Flags, die vielleicht nicht deutlich festgelegt sind.

LOKAL- ODER GLOBAL-VARIABLE

Ein Problem liegt darin, daß die in einem Makro verwendeten Variablen nur innerhalb dessen bekannt sind (d.h. sie sind eher "lokal" anstatt "global"). Dies kann häufig zu einer Verwirrung führen, ohne das entsprechende Vorteile vorhanden sind. Man sollte sich dieses Problem ständig bei der Verwendung von Makros vor Augen halten[1].

KOMMENTARE

Alle Assembler gestatten das Plazieren von Kommentaren in das Quellprogramm. Kommentare haben keinen Einfluß auf den Objektcode, aber sie helfen beim Lesen, Verstehen und Dokumentieren des Programms. Gute Kommentare sind ein wesentlicher Teil beim Schreiben von Assemblersprachen-Programmen. Ohne Kommentare sind Programme häufig sehr schwierig zu verstehen.

KOMMENTIER-TECHNIKEN

Die richtige Anwendung von Kommentaren, zusammen mit der Dokumentation, sollen in einem spätern Kapitel beschrieben werden, an dieser Stelle werden jedoch einige Richtlinien angegeben:

1) Verwenden Sie Kommentare um auszudrücken, welche Anwendungs-Aufgabe das Programm ausführt, nicht wie der Mikrocomputer die Befehle ausführt.
   Kommentare sollten etwa folgende Dinge angeben: "IST DIE TEMPERATUR OBERHALB DER GRENZE?", "ZEILEN-VORSCHUB ZUM FERNSCHREIBER" oder "PRÜFE LADESCHALTER".
   Kommentare sollten folgende Dinge nicht ausdrücken:
   "ADDIERE 1 ZUM AKKUMULATOR", "SPRINGE ZUM START" oder "PRÜFE ÜBERTRAG". Man sollte beschreiben, wie das Programm das System beeinflußt. Interne Einflüsse auf die CPU sind selten von Interesse.
2) Halten Sie Kommentare kurz und beschränken Sie sich auf das Wesentliche. Details sollten irgendwo anders in der Dokumentation zu finden sein.
3) Kommentieren Sie alle Schlüsselpunkte.
4) Kommentieren Sie nicht Standardbefehle oder Sequenzen, die Zähler und Zeiger ändern. Achten Sie besonders auf Befehle, die keine offensichtliche Bedeutung besitzen könnten.
5) Verwenden Sie keine obskuren Abkürzungen.
6) Machen Sie die Kommentare deutlich und lesbar.
7) Kommentieren Sie alle Definitionen und beschreiben ihren Zweck. Markieren Sie auch alle Tabellen und Datenspeicher-Bereiche.
8) Kommentieren Sie Abschnitte des Programms, sowie individuelle Befehle.
9) Verwenden Sie möglichst eine gleichmäßige Terminologie. Man kann (und sollte sogar) sich wiederholen. Die Schönheit der Sprache ist nicht wesentlich.
10) Bringen Sie Anmerkungen bei Punkten an, die zur Verwirrung führen könnten, z.B. "ÜBERTRAG WURDE VOM LETZTEN BEFEHL GESETZT". Man kann diese in der endgültigen Dokumentation weglassen.

Ein gut dokumentiertes Programm ist leicht zu verwenden. Die hierfür aufgewandte Zeit macht sich vielfach bezahlt. In den Programmbeispielen wird auf eine gute Kommentierung geachtet werden, obwohl für Lehrzwecke manchmal etwas überkommentiert werden wird.

ASSEMBLER-TYPEN

Obwohl alle Assembler dem gleichen Zweck dienen, variieren ihre Ausführungen in weitem Maße. Es soll nicht versucht werden, alle existierenden Typen von Assemblern zu beschreiben, Es sollen einfach die Ausdrücke definiert und erklärt werden.

CROSS-ASSEMBLER

Ein Cross-Assembler ist ein Assembler, der auf einem anderen Computer läuft, als auf jenem, für den er Programme assembliert.
Der Computer, auf dem der Cross-Assembler läuft, ist normalerweise ein großer Computer mit weitgehender Software-Unterstützung und schnellen Peripherie-Geräten, wie etwa eine IBM 360 oder 370, eine Univac 1108, oder eine Burroughs 6700. Der Computer, für den der Cross-Assembler Programme assembliert, ist normalerwise ein Mikroprozessor wie der 6502 oder 8080. Die meisten Cross-Assembler sind in FORTRAN geschrieben, so daß sie übertragbar sind.

RESIDENTE ASSEMBLER

Ein Selbst-Assembler oder residenter Assembler ist ein Assembler, der auf dem Computer läuft, für den er Programme assembliert.
Der Selbst-Assembler benötigt etliche Speicher und Peripheriegeräte, und er kann ziemlich langsam laufen.

MAKRO-ASSEMBLER

MIKRO-ASSEMBLER

Ein Makro-Assembler ist ein Assembler, der die Definition von Sequenzen von Befehlen als Makros gestattet. Ein Mikro-Assembler oder Mikroprogramm-Assembler ist ein Assembler, der zum Schreiben von Mikroprogrammen verwendet wird, die den Befehlssatz eines Computers definieren. Mikroprogrammierung hat nichts speziell mit Mikrocomputern zu tun.[2 3]

META-ASSEMBLER

Ein Meta-Assembler ist ein Assembler, der zahlreiche unterschiedliche Befehlssätze handhaben kann. Der Anwender muß den speziellen verwendeten Befehlssatz definieren.

ONE-PASS-ASSEMBLER

Ein One-Pass-Assembler ist ein Assembler, der das Assemblersprachen-Programm nur einmal durchläuft. Die wesentliche Schwierigkeit bei einem One-Pass-Assembler liegt im Vorhandensein eines Befehls, der sich auf eine Markierung bezieht, die später im Quellprogramm aufscheint. Ein One-Pass-Assembler muß irgendeine Möglichkeit besitzen, diese Vorwärts-Referenzen zu lösen.

TWO-PASS-ASSEMBLER

Ein Two-Pass-Assembler ist ein Assembler, der das Assemblersprachen-Quellprogramm zweimal durchläuft. Beim ersten Mal sammelt und definiert der Assembler einfach alle Symbole. Das zweite Mal ersetzt er die Referenzen durch die tatsächlichen Definitionen. Der Zwei-Pass-Assembler hat keine Probleme mit Vorwärts-Referenzen, kann jedoch etwas langsam sein, wenn kein schneller Massenspeicher (wie ein Floppy-Disk) zur Verfügung steht. Sonst muß der Assembler das Programm zweimal von einem langsamen Eingabe-Gerät (wie einem Lochstreifen-Leser eines Fernschreibers) lesen. Die meisten Assembler für Mikroprozessoren benötigen zwei Durchläufe.

FEHLER

Assembler besitzen normalerweise Fehler-Mitteilungen, die häufig aus einzelnen codierten Buchstaben bestehen. Einige typische Fehler sind:

- Undefinierte Namen (häufig ein orthografischer Fehler oder eine vergessene Definition).
- Ungültige Zeichen (z.B. die Zahl 2 in einer Binärzahl).
- Ungültiges Format (falsches Begrenzungszeichen oder unkorrekte Operanden).
- Ungültiger Ausdruck (z.B. zwei Operanden in einer Zeile).
- Ungültiger Wert (gewöhnlich zu groß).
- Fehlender Operand.
- Doppelte Definition (d.h. zwei verschiedene Werte wurden dem gleichen Namen zugewiesen).
- Ungültige Markierung (z.B. eine Markierung für eine Pseudo-Operation, die keine besitzend darf).
- Fehlende Markierung.
- Undefinierter Operationscode.

Bei der Interpretierung von Assembler-Fehlern muß man sich daran erinnern, daß der Assembler die falsche Spur verfolgen kann, wenn er einen vereinzelten Buchstaben, einen zusätzlichen Zwischenraum oder ein unkorrektes Satzzeichen findet. Zahlreiche Assembler werden dann fortfahren, die folgenden Befehle falsch zu interpretieren und bedeutungslose Fehler-Mitteilungen liefern. Es muß immer der erste Fehler sorgfältig untersucht werden. Darauf folgende können davon abhängen. Sorgfältige und ständige Verwendung von Standard-Formaten wird von vorneherein zahlreiche lästige Fehler vermeiden.

LADER

Der Lader ist das Programm, das die Ausgabe (Objektcode) tatsächlich vom Assembler nimmt und in den Speicher plaziert. Lader reichen von sehr einfachen bis zu sehr komplexen Ausführungen. Es sollen einige wenige Arten beschrieben werden.

BOOTSTRAP-LADER

Ein Bootstrap-Lader (oder Urlader) ist ein Programm, das nur einige seiner ersten eigenen Befehle benützt, um den Rest selbst zu laden oder ein anderes Laderprogramm in den Speicher zu bringen.
Der Urlader kann in einem ROM liegen, oder man muß ihn in den Computerspeicher durch Verwendung der Frontplatten-Schalter eingeben. Der Assembler kann einen Urlader an den Beginn des Objektprogrammes, das er erzeugt, plazieren.

RELOKATIBLER LADER

Ein relokatibler (verschiebbarer) Lader kann Programme an jede beliebige Stelle in den Speicher stellen. Er lädt normalerweise jedes Programm in den Speicherraum, der unmittelbar nach dem vom vorhergehenden Programm verwendeten folgt. Die Programme müssen jedoch imstande sein, dies selbst durchzuführen, d.h. sie müssen relokatibel sein. **Ein absoluter Lader wird dagegen die Programme immer in den gleichen Speicherbereich legen).**

BINDE-LADER

Ein Binde-Lader (linking loader) lädt Programme und Unterprogramme getrennt. Er liefert Quer-Referenzen, d.h. ein Befehl in einem Programm oder Unterprogramm bezieht sich auf eine Markierung in einem anderen. Objektprogramme, die durch einen Binde-Lader geladen werden, müssen in einem Assembler erzeugt werden, der externe Referenzen gestattet. Eine andere Lösung besteht in der Trennung der Binde- und Ladefunktionen und Ausführungen des Verbindens mittels eines Programmes, das Link-Editor genannt wird.

LITERATUR

1) A complete monograph on macros is M. Campbell-Kelly, "An Introduction to Macros," American Elsevier, New York, 1973.
2) A. Osborne, "Einführung in die Mikrocomputer-Technik" 3. Aufl. TEWI-Verlag, München.
3) A. K. Agrawala and T. G. Rauscher, Foundations of Microprogramming, Academic Press, New York, 1976.
4) D. W. Barron, "Assemblers and Loaders," American Elsevier, New York, 1972.
5) C. W. Gear, Computer Organization and Programming, McGraw-Hill, New York, 1974.

Kapitel 3
DER BEFEHLSSATZ DES 6502 IN ASSEMBLERSPRACHE

Wir sind nun soweit, daß wir mit der Schaffung von Programmen in Assemblersprache starten können. Wir beginnen in diesem Kapitel mit der Definition der individuellen Befehle des 6502 in Assemblersprache sowie mit den Syntaxregeln des MOS-Technology-Assemblers.

Es werden keinerlei Aspekte der Mikrocomputer-Hardware, Signale, Interfaces oder CPU-Architektur in diesem Buch besprochen. Diese Informationen sind detailliert in dem Buch "An Introduction to Microcomputers: Volume 2 – Some Real Microprocessors und Volume 3 – Some Real Support Devices".

In diesem Buch werden die Programmier-Techniken vom Standpunkt des Programmierers in Assemblersprache betrachtet, wobei Anschlüsse und Signale völlig irrelevant sind und es keine wesentlichen Unterschiede zwischen einem Minicomputer und einem Mikrocomputer gibt.

Unterbrechungen, direkter Speicherzugriff und die Stapel-Architektur des 6502 werden in späteren Kapiteln in diesem Buch beschrieben, in Verbindung mit der Besprechung der Assemblersprachen-Programmierung der gleichen Themen.

Dieses Kapitel enthält eine ausführliche Definition jedes Befehls in Assemblersprache.

Der ausführlichen Beschreibung der individuellen Befehle geht eine allgemeine Beschreibung des Befehlssatzes des 6502 voraus, wobei die Befehle in Gruppen eingeteilt werden: Die allgemein verwendeten (Tabelle 3-2) und die selten verwendeten (Tabelle 3-3) Befehle. Wenn Sie bereits ein erfahrener Assemblersprachen-Programmierer sind, ist diese Einteilung nicht besonders wichtig. Sind Sie jedoch ein Neuling in der Assemblersprachen-Programmierung, so ist es sehr empfehlenswert, mit dem Schreiben von Programmen nur unter Verwendung der "gebräuchlichsten" Befehle zu beginnen. Sobald Sie die Grundlagen der Assemblersprachen-Programmierung beherrschen, können Sie andere Befehle untersuchen und sie entsprechend einsetzen.

Tabelle 3-1. Häufig verwendete Befehle des 6502

Befehls-code	Bedeutung
ADC	Add with Carry (Addiere mit Übertrag)
AND	Logical AND (UNDiere logisch)
ASL	Arithmetic Shift Left (Verschiebe arithmetisch nach links)
BCC	Branch if Carry Clear (Verzweige, wenn Übertrag gelöscht)
BCS	Branch if Carry Set (Verzweige, wenn Übertrag gesetzt)
BEQ	Branch if Equal to Zero (Z = 1) (Verzweige, wenn gleich Null)
BMI	Branch if Minus (S = 1) (Verzweige, wenn Minus)
BNE	Branch if Not Equal to Zero (Z = 0) (Verzweige, wenn nicht gleich Null)
BPL	Branch if Plus (S = 0) (Verzweige, wenn Plus)
CMP	Compare Accumulator to Memory (Vergleiche Akkumulator mit Speicher)
DEC	Decrement (by 1) (Dekrementiere um 1)
DEX (DEY)	Decrement Index Register X (Y) by 1 (Dekrementiere Index-Register X (Y) um 1)
INC	Increment (by 1) (Inkrementiere um 1)
INX (INY)	Increment Index Register X (Y) by 1 (Inkrementiere Index-Register X (Y) um 1)
JMP	Jump to New Location (Springe zu neuem Speicherplatz)
JSR	Jump to Subroutine (Springe zu Unterprogramm)
LDA	Load Accumulator (Lade Akkumulator)
LDX (LDY)	Load Index Register X (Y) (Lade Index-Register X (Y))
LSR	Logical Shift Right (Verschiebe logisch nach rechts)
PHA	Push Accumulator onto Stack (Bringe Akkum. auf Stapel)
PLA	Pull Accumulator from Stack (Hole Akkum. vom Stapel)
ROL	Rotate Left through Carry (Rotiere links durch Übertrag)
ROR	Rotate Right through Carry (Rotiere rechts durch Übertrag)
RTS	Return from Subroutine (Kehre von Unterprogr. zurück)
SBC	Subtract with Borrow (Subtrahiere mit Borgen)
STA	Store Accumulator (Speichere Akkumulator)
STX (STY)	Store Index Register X (Y) (Speichere Index-Register X (Y))

Tabelle 3-2. Gelegentlich verwendete Befehle des 6502

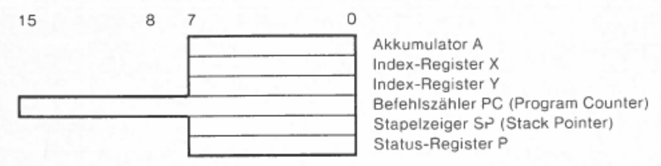
Befehls-code	Bedeutung
BIT	Bit Test (Bit-Test)
BRK	Break (Erzwinge Unterbrechung)
CLC	Clear Carry (Lösche Übertrag)
CLD	Clear Decimal Mode (Lösche Dezimal-Betriebsart)
CLI	Clear Interrupt Mask (Enable Interrupts) (Lösche Unterbrechungs-Maske (Gib Unterbrechungen frei))
CPX (CPY)	Compare with Index Register X (Y) (Vergleiche mit Index-Register X (Y))
EOR	Logical Exclusive-OR (Exklusiv-ODERiere logisch)
NOP	No Operation (Keine Operation)
ORA	Logical (Inclusive) OR (ODERiere logisch)
RTI	Return from Interrupt (Kehre von Unterbrechung zurück)
SEC	Set Carry (Setze Übertrag)
SED	Set Decimal Mode (Setze Dezimal-Betriebsart)
SEI	Set Interrupt Mask (Disable Interrupts) (Setze Unterbrechungs-Maske (Sperre Unterbrechungen))
TAX (TAY)	Transfer Accumulator to Index Register X (Y) (Transferiere Akkumulator zum Index-Register X (Y))
TXA (TYA)	Transfer Index Register X (Y) to Accumulator (Transferiere Index-Register X (Y) zum Akkumulator)

Tabelle 3-3. Selten verwendete Befehle des 6502.

Befehls-code	Bedeutung
BVC	Branch if Overflow Clear (Verzweige, wenn Überlauf gelöscht)
BVS	Branch if Overflow Set (Verzweige, wenn Überlauf gesetzt)
CLV	Clear Overflow (Lösche Überlauf)
PHP	Push Status Register onto Stack (Bringe Status-Register zum Stapel)
PLP	Pull Status Register from Stack (Hole Status-Register von Stapel)
TSX	Transfer Stack Pointer to Index Register X (Transferiere Stapelzeiger zum Index-Register X)
TXS	Transfer Index Register X to Stack Pointer (Transferiere Index-Register X zum Stapelzeiger)

CPU-REGISTER UND STATUS-FLAGS

Der Mikroprozessor 6502 besitzt einen Akkumulator, ein Status- (oder P-) Register, zwei Indexregister, einen Stapelzeiger und einen Befehlszähler. Diese Register können wie folgt dargestellt werden:



Das Statusregister des 6502 enthält sechs Status-Flags und ein Unterbrechungs-Steuerbit. Die sechs Status-Flags lauten:

Carry(C)	= Übertrags-Flag
Zero (Z)	= Null-Flag
Overflow(V)	= Überlauf-Flag
Negativ (N)	= Negativ-Flag (Vorzeichen)
Decimal Mode (D)	= Dezimale Betriebsart
Break (B)	= Unterbrechung

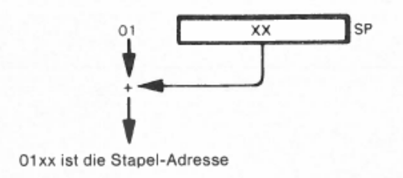
Die Status werden bestimmten Bit-Positionen innerhalb des Statusregisters wie folgt zugeordnet:



Der Akkumulator (A) ist ein primärer Akkumulator, wie er in "Einführung in die Mikroprozessor-Technik" beschrieben wurde.

Die Indexregister (X und Y) sind nur acht Bits lang, und somit anders als die typischen Mikrocomputer-Indexregister, wie sie in "Einführung in die Mikrocomputer-Technik" beschrieben sind. Sie ähneln mehr den klassischen Computer-Indexregistern, wie sie zur Aufbewahrung von Indizes, kurzen Versetzungen oder Zählern verwendet werden.

Der 6502 besitzt einen Stapel, der im Speicher eingerichtet und mittels des Stapelzeigers indiziert wird, wie in dem bereits erwähnten Buch beschrieben ist. **Der einzige Unterschied gegenüber diesen Beschreibungen besteht darin, daß der Stapelzeiger des 6502 nur acht Bit breit ist,** was bedeutet, daß die maximale Stapellänge 256 Bytes beträgt. Die CPU setzt immer 01_{16} als das höherwertige Byte jeder Stapel-Adresse ein, **was bedeutet, daß die Speicherplätze 0100_{16}** bis $01FF_{16}$ ständig dem Stapel zugewiesen werden:



01xx ist die Stapel-Adresse

Der kürzere Stapelzeiger des 6502 besitzt keine besondere Bedeutung, wenn Sie diese CPU als selbständiges Produkt verwenden. Ein Stapel mit 256 Bytes ist gewöhnlich ausreichend für jede typische Mikrocomputer-Anwendung. Und seine Lage im unteren Speicher bedeutet einfach, daß die niedrigen Speicher-Adressen als Lese/Schreib-Speicher eingerichtet werden müssen.

Der Befehlszähler des 6502 ist ein typischer Befehlszähler, wie er in der "Einführung in die Mikrocomputer-Technik" beschrieben wird.

Das Übertrags-Flag (Carry flag) bewahrt die Überträge des höchstwertigen Bits in jeder arithmetischen Operation auf. Das Übertrags-Flag ist auch in den Schiebe- und Rotations-Befehlen enthalten. **Die einzige ungewöhnliche Eigenschaft des Übertrags-Flags des 6502 besteht darin, daß es eine entgegengesetzte Bedeutung in Subtraktions-Operationen besitzt.** Nach einem SBC-Befehl wird der Übertrag gelöscht, wenn ein »Borgen« erforderlich war und gesetzt, wenn kein Borgen nötig war. Beachten Sie auch, daß der SBC-Befehl (Subtract with Carry) resultiert in $(A) = (A) - (M) - (1 - C)$, wobei M der andere Operand ist. Diese Verwendung ist anders als bei den meisten Mikroprozessoren oder Computern jüngster Herkunft, und der Anwender sollte dies beachten.

Das Null-Statusflag (Zero) ist Standard. Es wird auf 1 gesetzt, wenn irgendeine arithmetische oder logische Operation ein Resultat mit dem Wert 0 erzeugt. Es wird auf 0 gesetzt, wenn irgendeine arithmetische oder logische Operation ein Ergebnis verschieden von Null erzeugt.

Das Negativ-Statusflag (Negative) ist Standard. Es wird den Wert des hochwertigen (Vorzeichen-) Bits jedes arithmetischen oder logischen Resultats annehmen. Daher identifiziert ein Negativ-Statuswert von 1 ein negatives Ergebnis, und ein Negativ-Wert von 0 zeigt ein positives Ergebnis an. Das Negativ-Statuszeichen wird gesetzt oder gelöscht unter der Annahme, daß Sie binäre Arithmetik mit Vorzeichen verwenden. Wenn Sie keine Arithmetik mit Vorzeichen verwenden, können Sie den Negativ-Status ignorieren, oder Sie können ihn zur Identifizierung des Wertes des hochwertigen Bits des Ergebnisses einsetzen.

Das Dezimalbetriebs-Flag veranlaßt, wenn es gesetzt ist, die Befehle Add-with-Carry (Addiere mit Übertrag) und Subtract-with-Carry (Subtrahiere mit Übertrag) zur Ausführung von BCD-Operationen. Wenn daher der Dezimal-Betriebs-Status gesetzt ist und ein Add-with-Carry- oder Subtract-with-Carry-Befehl ausgeführt wurde, nimmt die CPU an, daß beide ursprünglichen 8-Bit-Werte gültige BCD-Zahlen sind und das erzeugte Ergebnis ebenso eine gültige BCD-Zahl sein wird. Da die CPU des 6502 dezimale Addition und Subtraktion ausführt, besteht kein Bedarf nach einem Halb-Übertrags- oder Zwischen-Status (Half-Carry status). Dieser Status ist in "Einführung in die Mikrocomputer-Technik" beschrieben. Ein Problem beim 6502 besteht darin, daß die gleiche Befehls-Sequenz verschiedene Resultate ergeben wird, abhängig davon, ob der Dezimalbetriebs-Status gesetzt oder gelöscht war. Daher können Verwirrung und Fehler auftreten, wenn das Dezimalbetriebs-Flag zufällig einen falschen Wert erhalten hat.

Der Break-Status gehört zu den Software-Unterbrechungen. **Wenn eine Software-Unterbrechung (BRK-Befehl) ausgeführt wird, wird die CPU-Logik des 6502 das Break-Statusflag setzen.**

I ist ein Standard-Haupt-Unterbrechungs-Freigabe/Sperr- oder Unterbrechungsmasken-Flag. Wenn I gleich 1 ist, werden die Unterbrechungen gesperrt. Ist I gleich 0, werden Unterbrechungen freigegeben.

Der Überlauf-Status (Overflow) ist ein typischer Überlauf mit Ausnahme, daß er als Steuer-Eingang eines 6502-Mikroprozessors verwendet werden kann. Erinnern Sie sich daran, daß bei Verwendung von Binär-Arithmetik der Überlauf-Status ein Ergebnis einer Größe anzeigt, die zu groß zur Darstellung in einer gegebenen Wortgröße ist. Der Überlauf-Status wurde detailliert in "Einführung in die Mikrocomputer-Technik" besprochen. Er gleicht dem Exklusiv-ODER von Überträgen aus den Bits 6 und 7 während arithmetischer Operationen. Der Mikroprozessor 6502 gestattet externer Logik das Setzen des Überlauf-Status und kann in diesem Fall anschließend als allgemeiner Logik-Indikator verwendet werden. Sie müssen sehr sorgfältig vorgehen, wenn Sie den Überlauf-Status auf diese Weise verwenden, da das gleiche Status-Flag durch arithmetische Befehle modifiziert wird. Sie als Programmierer müssen dafür sorgen, daß ein Befehl, der den Überlauf-Status modifiziert, nicht in der Zeit ausgeführt wird, in der externe Logik diesen Status setzt und die Programmlogik ihn hierauf testet.

UNTERSCHIEDE IN DER SCHREIBWEISE

Die Literatur des 6502 bezeichnet das Vorzeichen-Bit als Negativ-Bit mit dem Symbol N. In dem bereits erwähnten Buch "Einführung in die Mikrocomputer-Technik" wird dagegen als Standardsymbol für das Vorzeichen das Zeichen S (Sign bit) verwendet. In diesem Buch werden wir wir jedoch die allgemein eingeführte Bezeichnung N verwenden.

SPEICHER-ADRESSIERUNGSARTEN DES 6502

Der 6502 bietet elf grundlegende Adressier-Arten:

1) Speicher – unmittelbar
2) Speicher – absolut oder direkt, Nicht-Null-Seite (non-zero-page)
3) Speicher – Nullseite (direkt)
4) Impliziert oder inhärent
5) Akkumulator
6) Vor-indiziert indirekt
7) Nach-indiziert indirekt
8) Nullseite, indiziert (auch Basis-Seite, indiziert, genannt)
9) Absolut indiziert
10) Relativ
11) Indirekt

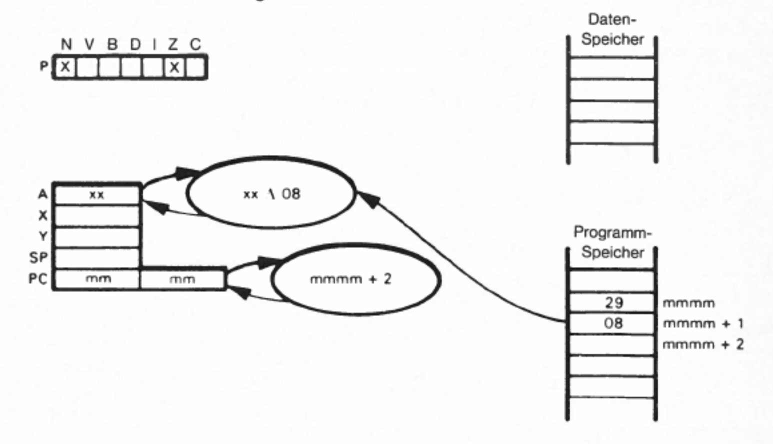
Es gibt wesentliche Variationen in den Ausdrücken, welche Methoden mit welchen Befehlen zulässig sind. Siehe Tabelle 3-4 für die mit jedem Befehl verfügbaren Adressier-Optionen.

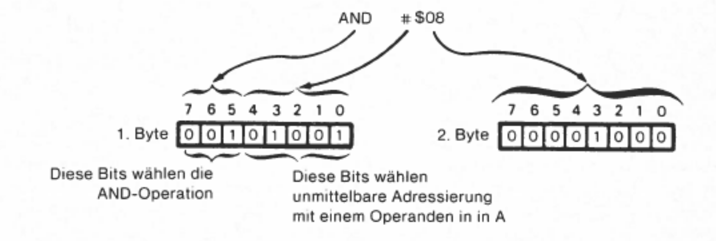
Speicher – Unmittelbar

Bei dieser Form der Adressierung liegt einer der Operanden in dem Byte vor, das unmittelbar dem Byte des Objektcodes folgt. Ein unmittelbarer Operand wird durch Vorsetzen des Symbols # vor dem Operanden spezifiziert. Zum Beispiel fordert

AND #$08

den Assembler auf, einen Befehl zu erzeugen, der den Wert von 08_{16} mit dem Inhalt des Akkumulators logisch UNDieren wird.



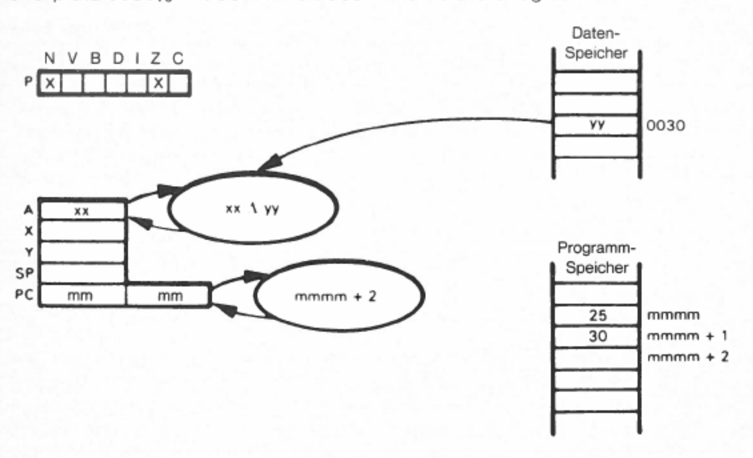


Speicher – direkt

Diese Art der Adressierung verwendet das zweite – oder zweite und dritte (wenn nicht auf der Null- oder Basis-Seite) – Byte des Befehls zur Identifizierung der Adresse eines Operanden im Speicher. Die Nullseiten-Version wird spezifiert, wenn der im Befehl als Operand verwendete Ausdruck auf einen Wert zwischen 00_{16} und FF_{16} zurückgeht. Zum Beispiel

AND $30

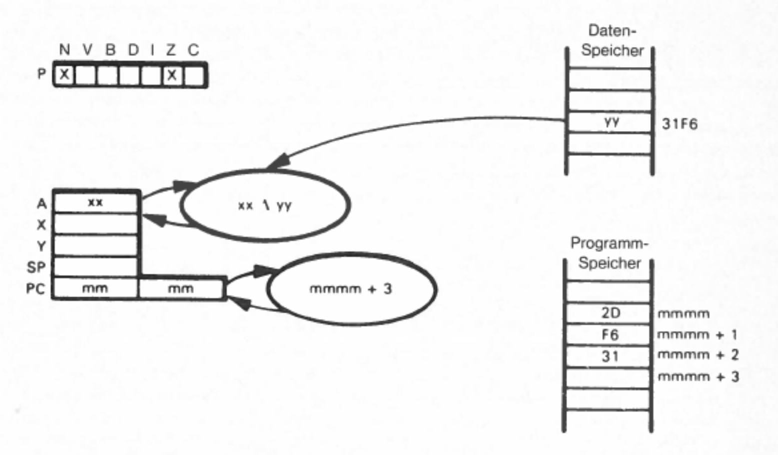
fordert den Assembler auf, einen UND-Befehl zu erzeugen, der den Wert im Speicherplatz 0030_{16} mit dem Inhalt des Akkumulators logisch UNDiert.



Die Nicht-Null-Seiten- (absolute) Version ist ähnlich, mit der Ausnahme, daß die Adresse des Operanden zwei Bytes belegt. Zum Beispiel

AND $31F6

fordert den Assembler auf, einen UND-Befehl zu erzeugen, der den Wert im Speicherplatz 0030_{16} mit dem Inhalt des Akkumulators logisch UNDiert.



SPEICHERN VON ADRESSEN

Sie sollten beachten, daß 16-Bit-Adressen mit den acht höchstwertigen Bits zuerst (bei der niedrigeren Adresse) gespeichert werden, gefolgt von den acht höchstwertigen Bits (in der höheren Adresse). Dies ist das gleiche Verfahren, wie es bei den Mikroprozessoren 8080, 8085 und Z80 verwendet wird, jedoch entgegengesetzt dem beim Mikroprozessor 6800 verwendeten Verfahren.

Implizierte oder inhärente Adressierung

Diese Betriebsart bedeutet, daß keine Adressen zur Ausführung des Befehls erforderlich sind. Typische Beispiele inhärenter Adressierung sind CLC (Clear Carry) und TAX (Transferiere Register A zu Register X).

Akkumulator-Adressierung

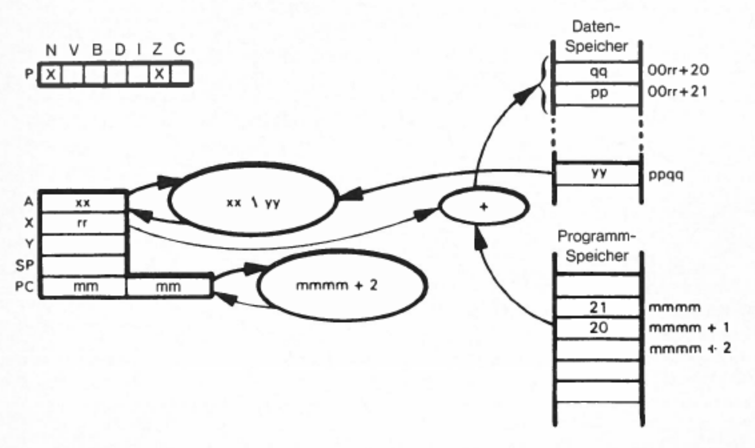
Diese Betriebsart bedeutet, daß der Befehl die Daten im Akkumulator verarbeitet. Beim Mikroprozessor 6502 sind die einzigen Akkumulator-Befehle die Verschiebungen ASL (Arithmetische Links-Verschiebung), LSR (Logische Rechtsverschiebung), ROL (Rotiere links durch Übertrag) und ROR (Rotiere rechts durch Übertrag).

Vor-indizierte indirekte Adressierung

Diese Betriebsart bedeutet, daß das zweite Byte des Befehls zum Inhalt des X-Indexregisters zum Zugriff auf einen Speicherplatz in den ersten 256 Bytes des Speichers addiert wird, wodurch die indirekte Adresse gefunden wird. Hier wird eine Addition verwendet, bei der jeder Übertrag, der bei einer Adressen-Addition gebildet wird, gelöscht wird. Zum Beispiel

AND ($20,X)

fordert den Assembler auf, einen Befehl zu erzeugen, der den Inhalt des Akkumulators mit dem Inhalt des Bytes, das durch den Speicherplatz auf der Nullseite adressiert wird, der durch die Summe von 20_{16} und dem Inhalt des Indexregisters gegeben ist, logisch UNDiert. Beachten Sie die Verwendung der Klammern im Adressenfeld zur Anzeige von "Inhalt von".



Erinnern Sie sich daran, daß der Übertrag aus der Adressen-Addition ignoriert wird, das heißt, die Adresse des ersten Adressen-Bytes ist ein Zahl in Mod 256. Beachten Sie, daß die indirekte Adresse mit ihren niedrigstwertigen Bits zuerst (bei der niedrigeren Adresse) gespeichert wird. Beachten Sie auch, daß die Adresse zwei Speicher-Bytes belegt.

Nur das X-Indexregister kann für vor-indizierte indirekte Adressierung verwendet werden.

Nach-indizierte indirekte Adressierung

Dieses Verfahren bedeutet, daß das zweite Byte des Befehls eine Adresse in den ersten 256 Speicherbytes enthält. Diese Adresse und der nächste Speicherplatz enthalten eine Adresse, die zum Inhalt des Y-Indexregisters zur Erlangung der effektiven Adresse addiert wird.

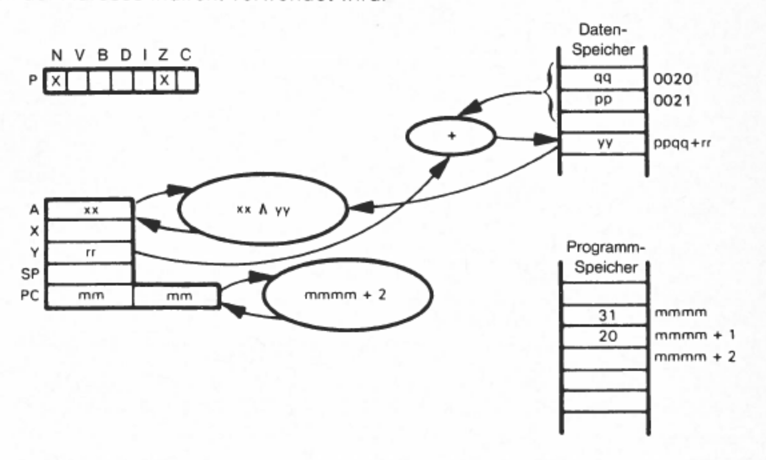
Beachten Sie die Unterschiede zwischen diesem Verfahren und der vor-indizierten indirekten Adressierung:

1) Bei vor-indizierter indirekter Adressierung wird die Indizierung vor der Adressierung ausgeführt, während bei der nach-indizierten Adressierung die Adressierung vor der Indizierung ausgeführt wird.
2) Vor-indizierte indirekte Adressierung verwendet das X-Indexregister, während nach-indizierte indirekte Adressierung das Y-Indexregister verwendet.
3) Vor-indizierte indirekte Adressierung ist nützlich bei der Auswahl eines Satzes von indirekt zu verwendenden Adressen, während nach-indizierte indirekte Adressierung für den Zugriff auf Elemente in einem Feld oder einer Tabelle geeignet ist, für die die Basis-Adresse indirekt erhalten worden ist.

Ein Beispiel für nach-indizierte indirekte Adressierung ist

AND ($20),Y

welches den Assembler auffordert, einen Befehl zu erzeugen, der den Inhalt des Akkumulators mit dem Inhalt des Bytes, adressiert durch Addition des Y-Indexregisters zur Adresse im Speicherplatz 0020_{16}, logisch UNDieren wird. Beachten Sie, daß hier nur $20 innerhalb der Klammern steht, da nur dieser Teil der Adresse indirekt verwendet wird.



Hier wird wiederum die indirekte Adresse mit ihrem niedrigstwertigen Byte zuerst (bei der niedrigeren Adresse) gespeichert. Anders als bei der vor-indizierten Adressierung ist diese Adressen-Addition eine volle 16-Bit-Addition. Jedoch wird jeder Übertrag von Bit 15 ignoriert. Es kann nur das Y-Indexregister mit nach-indizierter, indirekter Adressierung verwendet werden.

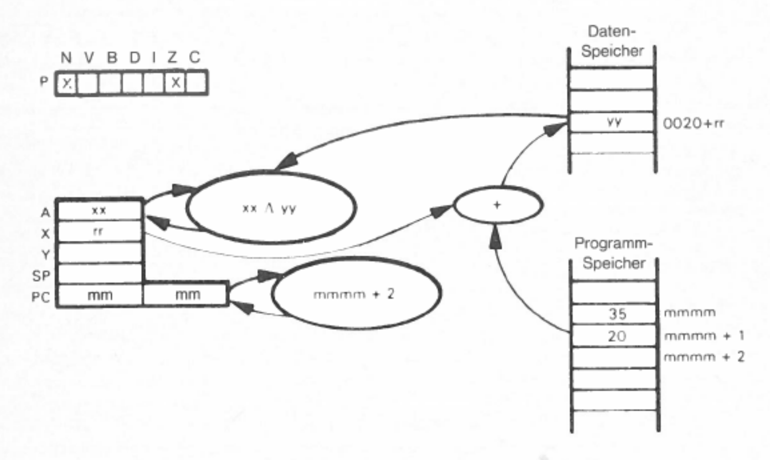
Indizierte Adressierung

Diese Art der Adressierung verwendet das zweite – oder zweite und dritte (wenn nicht auf der Nullseite) – Byte des Befehls zur Spezifizierung der Basis-Adresse. Diese Basis-Adresse wird dann zum Inhalt des Indexregisters X oder Y addiert, um die effektive Adresse zu erhalten. X und Y sind nicht austauschbar, da keine Befehle beide Formen der einfachen Indizierung mit sowohl X als auch Y besitzen. In der Tat ist der einzige Befehl, der eine Nullseiten-Indizierung mit Y gestattet, der Befehl LDX (Load Index Register X) und STX (Store Index Register X). Sie sollten sich auf die Tabelle 3-4 beziehen, um zu bestimmen, welche Adressier-Optionen bei Ihrem Befehl verfügbar sind.

Ein typisches Beispiel indizierter Nullseiten-Adressierung ist

AND $20,X

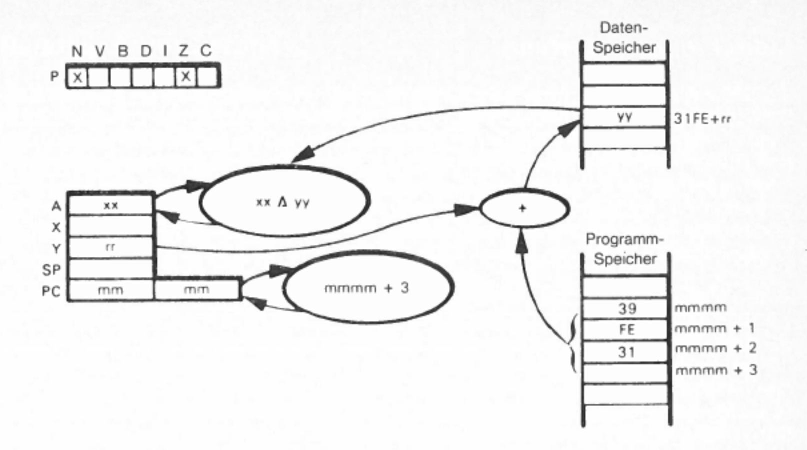
Hierdurch wird der Assembler aufgefordert, den Befehl zu bilden, der den Inhalt des Akkumulators mit dem Inhalt des Bytes an der Adresse, gegeben durch die Summe von 20_{16} und dem Inhalt des X-Indexregisters, logisch UNDiert. Dies ist ein 2-Byte-Befehl, da die Adresse innerhalb der ersten 256 Bytes des Speichers liegt. Beachten Sie, daß es keine 2-Byte-Form von AND $20,X gibt, obwohl eine allgemeinere 3-Byte-Form dieses Befehls existiert.



Ein typisches Beispiel absoluter indizierter Adressierung ist

AND $31FE,Y

welches den Assembler auffordert, den Befehl zu erzeugen, der den Inhalt des Akkumulators mit dem Inhalt des Bytes bei der Adresse, die durch die Summe von $31FE_{16}$ und dem Inhalt des Y-Indexregisters gegeben ist, logisch UNDieren wird. Dies ist ein 3-Byte-Befehl, da die Basis-Adresse nicht innerhalb der ersten 256 Bytes des Speichers liegt.



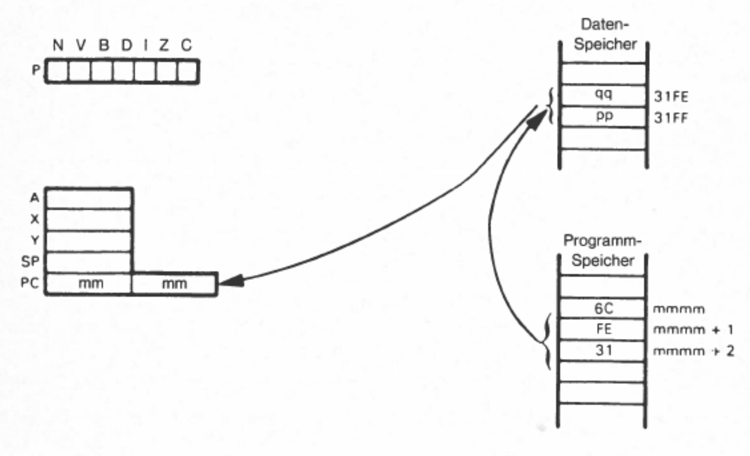
Hier kann das Indexregister X als auch das Indexregister Y verwendet werden. Jedoch einige Befehle (wie etwa ASL, DEC, INC, LSR, ROL und ROR) gestatten nur das Indexregister in dieser Betriebsart. Dies ist auch (mehr logisch) mit dem Befehl LDY (Load Index Register Y) und STY (Store Index Register Y) der Fall.

Indirekte Adressierung

Indirekte Adressierung ist nur bei dem Befehl JMP (Jump to New Location = Springe zu neuem Speicherplatz) möglich. Hierbei enthält das zweite und dritte Byte des Befehls die Adresse, bei der die effektive Adresse liegt. Beachten Sie, daß die indirekte Adresse jeden beliebigen Wert besitzen und überall im Speicher liegen kann. Offensichtlich kann diese Adressier-Art als ein spezieller Fall entweder der nach-indizierten indirekten Adressierung oder vor-indizierten indirekten Adressierung angesehen werden, bei der das Indexregister null enthält. Ein typisches Beispiel ist:

JMP ($31FE)

das den Assembler auffordert, einen JMP-Befehl zu bilden, der den Befehlszähler von dem Speicherplatz lädt, der durch den Inhalt der Speicherplätze $31FE_{16}$ und $31FF_{16}$ adressiert wird. Erinnern Sie sich daran, daß absolute Adressen 16 Bits lang sind und zwei Speicherbytes belegen. Die an einer Adresse liegenden Daten sind jedoch 8 Bits lang. Dieses Durcheinander betrifft alle 8-Bit-Prozessoren, stellt jedoch ein besonderes Problem beim 6502 infolge seiner zahlreichen indirekten und indizierten Adressierungsarten dar. Indirekte Adressierung wird ausführlicher in "Einführung in die Mikrocomputer-Technik", Kapitel 6, beschrieben. Erinnern Sie sich daran, daß alle Adressen mit ihrem niedrigstwertigen Byte zuerst (bei der niedrigeren Adresse) gespeichert werden.



Der endgültige Wert des Befehlszählers ist ppqq.

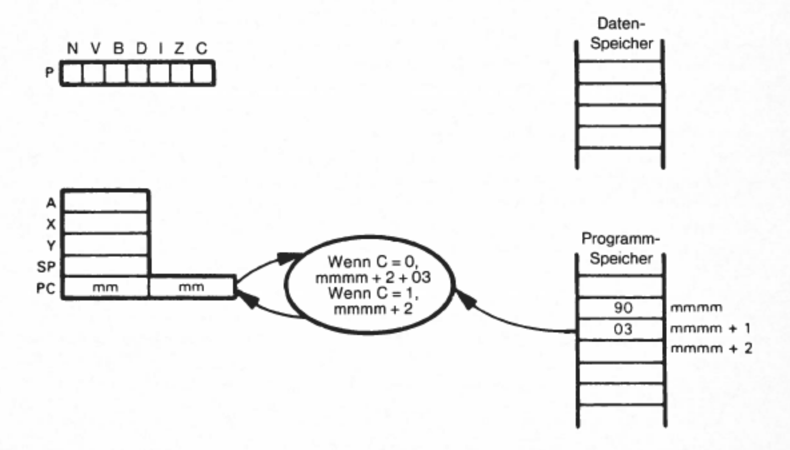
Relative Adressierung

Befehle für bedingte Verzweigungen verwenden programmabhängige Adressierung. Eine Einzelbyte-Versetzung wird als Binärzahl mit Vorzeichen behandelt, die zum Befehlszähler addiert wird, nachdem der Inhalt des Befehlszählers inkrementiert wurde, um den nächstfolgenden Befehl zu adressieren. Dies gestattet Versetzungen im Bereich von $+129_{10}$ bis -126_{10} Bytes. Ein typisches Beispiel ist

BCC *+5

welches den Assembler auffordert, einen Befehl BCC (Branch on Carry Clear, d.h. Verzweigung wenn Übertrag = 0) zu erzeugen, der den Befehlszähler mit seinem momentanen Wert plus fünf laden wird, wenn der Übertrag in Wirklichkeit Null ist. Wenn der Übertrag Eins ist, so führt der Befehl nichts aus. Beachten Sie, daß der Befehl selbst zwei Speicherbytes belegt und die Versetzung ab dem Ende des Befehls gemessen wird. Daher sollte die Versetzung 3 sein, um eine Verzweigung zum Speicherplatz fünf, gerechnet ab dem einen, in dem das erste Byte des Befehls liegt, zu bewirken. Beachten Sie, daß das Symbol * für den momentanen Wert des Befehlszählers verwendet wird (in Wirklichkeit der Stellenzähler des Assemblers wie in Kapitel 2 beschrieben).

Die Ausführung des Befehls BCC*+5 kann wie nachstehend dargestellt beschrieben werden. Beachten Sie, daß der gesamte Befehl vom Speicher geholt wird, bevor die Bestimmungsadresse berechnet wird. Beachten Sie auch, daß es keine anderen verfügbaren Adressier-Arten bei Befehlen für bedingte Adressierung gibt.



BEFEHLSSATZ DES 6502

Befehle erschrecken häufig Mikrocomputer-Anwender, die Neulinge in der Programmierung sind. Nimmt man die Befehle als getrennten Vorgang an, so sind die mit der Ausführung eines einzelnen Befehls verbundenen Operationen einfach zu überblicken. Der Zweck dieses Kapitels besteht im Trennen und Erklären von Operationen.

Weshalb nennt man die Befehle eines Mikrocomputers einen "Befehlssatz"? Der Grund hierfür ist, daß die Befehle vom Entwickler eines Mikrocomputers sehr sorgfältig ausgewählt wurden. Sie müssen auf einfache Weise komplexe Operationen als eine Folge einfacher Vorgänge ausführen, von denen jeder durch einen Befehl eines gut durchdachten "Befehlssatzes" dargestellt wird.

In Übereinstimmung mit "An Introduction to Microcomputers: Volume 2" zeigt die Tabelle 3-4 den Befehlssatz des Mikrocomputers 6502, wobei ähnliche Befehle in Gruppen zusammengefaßt sind. Individuelle Befehle werden in alphabetischer Reihenfolge des Objektcodes in Tabelle 3-5 aufgelistet und eine alphabetische Reihenfolge der Befehls-Mnemoniks in Tabelle 3-6. Tabelle 3-6 vergleicht auch den Befehlssatz des 6800 mit dem des 6502. Wir werden den 6800 und 6502 später in diesem Kapitel besprechen, nachdem wir uns eingehend mit dem Befehlssatz des 6502 befaßt haben.

Zusätzlich zur Feststellung, was jeder Befehl ausführt, wird in den Besprechungen der individuellen Befehle der Zweck des Befehls innerhalb der normalen Programmlogik erklärt.

ABKÜRZUNGEN

Folgende Abkürzungen werden in diesem Kapitel verwendet:

Die Register:

A	Akkumulator
X	Indexregister X
Y	Indexregister Y
PC	Befehlszähler (Program Counter)
SP	Stapelzeiger (Stack Pointer)
P	Statusregister, wobei die Bits folgendermaßen zugeordnet sind:

7	6	5	4	3	2	1	0	← Bit-Nummer
N	V		B	D	I	Z	C	← Status-Register (P)

Bit 5: Reserviert für Erweiterung (derzeit nicht verwendet)

Status (Flags):

N	Negativ-Status (Negativ- oder Vorzeichen-Status)
V	Overflow-Status (Überlauf-Status)
B	Break-Status
D	Decimal-Mode-Status (Dezimal-Betriebsart-Status)
I	Interrupt-Disable-Status (Unterbrechungs-Sperr-Status)
Z	Zero-Status (Null-Status)
C	Carry-Status (Übertrags-Status)

Symbole in der mit STATUS bezeichneten Spalte:

(leer)	Operation beeinflußt das Flag nicht
X	Operation beeinflußt Flag
0	Operation löscht Flag
1	Operation setzt Flag
6	Operation gibt Bit 6 des Speicherplatzes wieder
7	Operation gibt Bit 7 des Speicherplatzes wieder
addr	8 Bits der absoluten oder Basis-Adresse
[addr+1,addr]	Die Adresse, gebildet aus dem Inhalt der Speicherplätze addr und addr+1. Diese Adresse wird bei der nach-indizierten indirekten Adressierung verwendet.
addr16	16 Bits einer absoluten oder Basis-Adresse
data	8 Bits unmittelbarer Daten
disp	Eine 8-Bit-Adressenversetzung (displacement) mit Vorzeichen
label	Absolute Adresse mit 16 Bits, Ziel eines Befehls "Springe" oder "Springe zu Unterprogramm"
PC(HI)	Die hochwertigen 8 Bits des Befehlszählers (Program Counter)
PC(LO)	Die niederwertigen 8 Bits des Befehlszählers
pp	Das zweite Byte eines Befehlscodes eines Zwei- oder Drei-Byte-Befehls
qq	Das dritte Byte eines Drei-Byte-Objektcodes
[]	Inhalt des Speicherplatzes, der mit eckigen Klammern eingeschlossen ist. Beispielsweise stellt [FFFE] den Inhalt des Speicherplatzes $FFFE_{16}$ dar. [addr 16+X] stellt den Inhalt des Speicherplatzes dar, der durch Addition des Inhalts des Registers X zu addr16 adressiert wird. [SP] stellt den Wert an der Spitze des Stapels dar (Inhalt des Speicherplatzes, adressiert durch den Stapelzeiger)
[[ ]]	Indirekte Adressierung: Der Inhalt des Speicherbytes, adressiert durch den Inhalt des innerhalb der eckigen Klammer angegebenen Speicherplatzes. Zum Beispiel stellt [[addr+X]] den Inhalt eines Speicherplatzes dar, der über die vor-indizierte indirekte Adressierung adressiert wurde.
+	Addition – entweder Binär-Addition ohne Vorzeichen oder BCD-Addition, abhängig vom Zustand des Dezimal-Betriebsarts-Status.
−	Binäre oder BCD-Subtraktion, ausgeführt durch Addition des Zweierkomplements des Subtrahenden zum Minuenden.
—	Das Einerkomplement der unter dem Strich angegebenen Größe. Zum Beispiel stellt $\overline{A}$ das Komplement des Inhalts des Akkumulators dar. $\overline{C}$ stellt das Komplement des Wertes des Übertrags-Status (Carry-Status) dar.
∧	Logisch UND
∨	Logisch ODER
⊻	Logisch Exklusiv-ODER
←	Daten werden in der Richtung des Pfeiles transferiert.

BEFEHLS-MNEMONIKS

Tabelle 3-4 faßt den Befehlssatz des 6502 zusammen. Die BEFEHLS-Spalte zeigt die Befehls-Mnemoniks (LDA, STA, CLC) und die Operanden, falls welche vorliegen, die mit dem Befehls-Mnemonik verwendet werden.

Der feste Teil des Assemblersprachen-Befehls wird in GROSSBUCHSTABEN dargestellt. Der variable Teil (unmittelbare Daten, Adresse oder Markierung) wird in Kleinbuchstaben gezeigt.

Wenn ein Mnemonik mehr als eine Art eines Operanden hat, wird jede Art separat ohne Wiederholung des Mnemonik aufgelistet. Einige Beispiele sehen folgendermaßen aus:

```
            STX
               addr
               addr.Y
               addr16

wird zu:    STX $75
            STX $60.Y
            STX $4276
```

BEFEHLS-OBJEKTCODES

Für Befehlsbytes ohne Variationen werden die Objektcodes als zwei hexadezimale Ziffern dargestellt (z.B. 8A). Für Befehlsbytes mit Variationen wird der Objektcode mit acht Binärziffern dargestellt (z.B. 101aaa01).

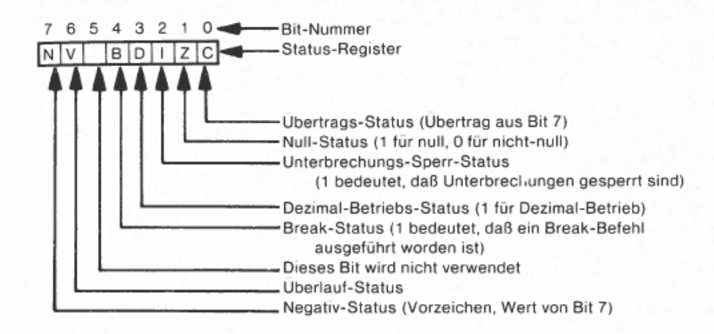
Der Objektcode und die Befehlslänge in Bytes sind in Tabelle 3-4 für jede Befehls-Variante gezeigt. Tabelle 3-5 listet den Objektcode in numerische Ordnung auf, und Tabelle 3-6 zeigt die entsprechenden Objektcodes für die Mnemoniks, aufgezählt in alphabetischer Reihenfolge.

BEFEHLS-AUSFÜHRUNGSZEITEN

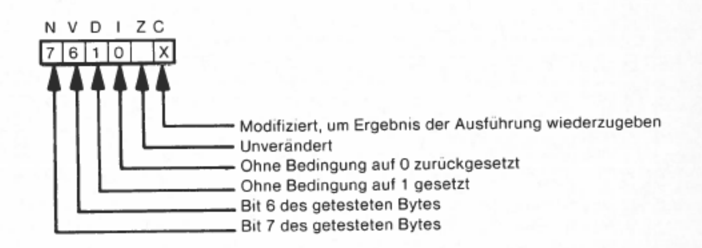
Tabelle 3-4 listet die Befehls-Ausführungszeiten in Anzahl der Taktperioden auf. Die tatsächliche Ausführungszeit kann durch Dividieren der gegebenen Zahl der Taktperioden durch die Taktgeschwindigkeit ermittelt werden. Beispielsweise ergibt sich für einen Befehl, der fünf Taktperioden benötigt, bei einem 2 MHz-Takt eine Ausführungszeit von 2.5 Mikrosekunden.

STATUS

Die Status-Flags sind im Statusregister (P) wie folgt gespeichert:



Bei der Beschreibung der individuellen Befehle wird der Einfluß der Befehls-Ausführung auf die Status wie folgt gezeigt:



STATUS-ÄNDERUNGEN BEI BEFEHLS-AUSFÜHRUNG

Ein X zeigt einen Status an, der gesetzt oder gelöscht wird. Ein 0 identifiziert einen Status, der immer gelöscht wird. Eine 1 identifiziert einen Status, der immer gesetzt wird. Eine leere Stelle bedeutet, daß der Status nicht geändert wird. Die Zahlen 7 und 6 zeigen, daß das Flag den Wert von Bit 7 oder Bit 6 des Bytes enthält, das durch den Befehl getestet wird.

Tabelle 3-4. Eine Zusammenfassung des 6502-Befehlssatzes

Typ	Befehl	Objekt-Code	Byte	Takt-Perioden	Status N	V	D	I	Z	C	Ausgeführte Operation	
E/A und primäres Speicher-Ansprechen	LDA										Lade Akkumulator vom Speicher.	
	addr	A5 pp	2	3	X				X		A←[addr]	Null-Seite direkt
	addr,X	B5 pp	2	4	X				X		A←[addr+X]	Null-Seite indiziert
	(addr,X)	A1 pp	2	6	X				X		A←[[addr+X]]	Vor-indiziert indirekt
	(addr),Y	B1 pp	2	5*	X				X		A←[[addr+1,addr]+Y]	Nach-indiziert indirekt
	addr16	AD ppqq	3	4	X				X		A←[addr16]	Erweitert direkt
	addr16,X od. Y	11011x01 ppqq	3	4*	X				X		A←[addr16+X] oder A←[addr16+Y]	Absolut indiziert
	STA										Speichere Akkumulator in Speicher	
	addr	85 pp	2	3							[addr]←A	Null-Seite direkt
	addr,X	95 pp	2	4							[addr+X]←A	Null-Seite indiziert
	(addr,X)	81 pp	2	6							[[addr+X]]←A	Vor-indiziert indirekt
	(addr),Y	91 pp	2	6							[[addr+1,addr]+Y]←A	Nach-indiziert indirekt
	add16	8D ppqq	3	4							[addr16]←A	Erweitert direkt
	addr16,X od. Y	10011x01 ppqq	3	5							[addr16+X]←A oder [addr16+Y]←A	Absolut indiziert
	LDX										Lade Index-Register X vom Speicher. Index nur über Register Y.	
	addr	A6 pp	2	3	X				X		X←[addr]	Null-Seite direkt
	addr,Y	B6 pp	2	4	X				X		X←[addr+Y]	Null-Seite indiziert
	addr16	AE ppqq	3	4	X				X		X←[addr16]	Erweitert indiziert
	addr16,Y	BE ppqq	3	4*	X				X		X←[addr16+Y]	Absolut indiziert
	STX										Speichere Index-Reg. X zum Speicher. Index nur über Reg. Y.	
	addr	86 pp	2	3							[addr]←X	Null-Seite direkt
	addr,Y	96 pp	2	4							[addr+Y]←X	Null-Seite indiziert
	addr16	8E ppqq	3	4							[addr16]←X	Erweitert direkt
	LDY										Lade Index-Register Y vom Speicher. Index nur über Register X.	
	addr	A4 pp	2	3	X				X		Y←[addr]	Null-Seite direkt
	addr,X	B4 pp	2	4	X				X		Y←[addr+X]	Null-Seite indiziert
	addr16	AC ppqq	3	4	X				X		Y←[addr16]	Erweitert direkt
	addr16,X	BC ppqq	3	4*	X				X		Y←[addr16+X]	Absolut indiziert

*Addiere eine Taktperiode, wenn eine Seiten-Grenze überschritten wird.
Im Objekt-Code zeigt "X" das Index-Register an: X=0 für Register Y, X=1 für Register X.

Tabelle 3-4. Eine Zusammenfassung des 6502-Befehlssatzes (Fortsetzung)

Typ	Befehl	Objekt-Code	Byte	Takt-Perioden	Status N	V	D	I	Z	C	Ausgeführte Operation	
E/A u. primäres Speicher-Ansprechen (Forts.)	STY										Speichere Index-Register Y zum Speicher. Index nur über Register X.	
	addr	84 pp	2	3							[addr]←Y	Null-Seite direkt
	addr,X	94 pp	2	4							[addr+X]←Y	Null-Seite indiziert
	addr16	8C ppqq	3	4							[addr16]←Y	Erweitert direkt
Sekundäres Speicher-Ansprechen (Speicher-Operation)	ADC										Addiere Inhalt des Speicherplatzes mit Übertrag zu jenem des Akkumul.	
	addr	65 pp	2	3	X	X			X	X	A←A+[addr]+C	Null-Seite direkt
	addr,X	75 pp	2	4	X	X			X	X	A←A+[addr+X]+C	Null-Seite indiziert
	(addr,X)	61 pp	2	6	X	X			X	X	A←A+[[addr+X]]+C	Vor-indiziert indirekt
	(addr),Y	71 pp	2	5*	X	X			X	X	A←A+[[addr+1,addr]+Y]+C	Nach-indiziert indirekt
	add16	6D ppqq	3	4	X	X			X	X	A←A+[addr16]+C	Erweitert direkt
	addr16,X od. Y	01111x01 ppqq	3	4*							A←A+[addr16+X]+C od. A←A+[addr16+Y]+C (Null-Flag ist in der Dezimal-Betriebsart nicht gültig)	Absolut indiziert
	AND										UNDiere Inhalt des Akkumulators mit jenem des Speicherplatzes.	
	addr	25 pp	2	3	X				X		A←A ∧ [addr]	Null-Seite direkt
	addr,X	35 pp	2	4	X				X		A←A ∧ [addr+Y]	Null-Seite indiziert
	(addr,X)	21 pp	2	6	X				X		A←A ∧ [[addr+X]]	Vor-indiziert indirekt
	(addr),Y	31 pp	2	5*	X				X		A←A ∧ [[addr+1, addr]+Y	Nach-indizirt indirekt
	addr16	2D ppqq	3	4	X				X		A←A ∧ [addr16]	Erweitert indiziert
	addr16,X od. Y	00111x01 ppqq	3	4*	X				X		A←A ∧ [addr16+X]od.A←A ∧ [addr16+Y]	Absolut indiziert
	BIT										UNDiere Inhalt des Akkumulators mit jenem des Speicherplatzes. Es werden nur die Status-Bits beeinflußt.	
	addr	24 pp	2	3	7	6			X		A ∧ [addr]	Null-Seite direkt
	addr16	2C ppqq	3	4	7	6			X		A ∧ [addr16]	Erweitert direkt

*Addiere eine Taktperiode, wenn eine Seiten-Grenze überschritten wird.
Im Objekt-Code zeigt "X" das Index-Register an: X=0 für Register Y, X=1 für Register X.

Tabelle 3-4. Eine Zusammenfassung des 6502-Befehlssatzes (Fortsetzung)

Typ	Befehl	Objekt-Code	Byte	Takt-Perioden	Status N	V	D	I	Z	C	Ausgeführte Operation	
Sekundäres Speicher-Ansprechen (Speicher-Operation) (Fortsetzung)	CMP										Vergleiche den Inhalt des Akkumulators mit jenem des Speicherplatzes. Nur die Status-Bits werden beeinflußt.	
	addr	C5 pp	2	3	X				X	X	A–[addr]	Nullseite direkt
	addr,X	D5 pp	2	4	X				X	X	A–[addr+X]	Nullseite indiziert
	(addr,X)	C1 pp	2	6	X				X	X	A–[[addr+X]]	Vorindiziert indirekt
	(addr),Y	D1 pp	2	5*	X				X	X	A–[[addr+1, addr]+Y]	Nachindiziert indirekt
	addr16	CD ppqq	3	4	X				X	X	A–[addr16]	Erweitert direkt
	addr16,X od. Y	11011x01 ppqq	3	4*	X				X	X	A–[addr16+X] oder A–[addr16+Y]	Absolut indiziert
	EOR										Exklusiv-ODERiere den Inhalt des Akkumulators mit jenem des Speicherplatzes.	
	addr	45 pp	2	3	X				X		A←A$\veebar$[addr]	Nullseite direkt
	addr,X	55 pp	2	4	X				X		A←A$\veebar$[addr+X]	Nullseite indiziert
	(addr,X)	41 pp	2	6	X				X		A←A$\veebar$[[addr+X]]	Vorindiziert indirekt
	(addr),Y	51 pp	2	5*	X				X		A←A$\veebar$[[addr+1, addr]+Y]	Nachindiziert indirekt
	addr16	4D ppqq	3	4	X				X		A←A$\veebar$[addr16]	Erweitert direkt
	addr16,X od. Y	01011x01 ppqq	3	4*	X				X		A←A$\veebar$[addr16+X] od. A←A$\veebar$[addr16+Y]	Absolut indiziert
	ORA										ODERiere den Inhalt des Akkumulators mit jenem des Speicherplatzes.	
	addr	05 pp	2	3	X				X		A←AV[addr]	Nullseite direkt
	addr,X	15 pp	2	4	X				X		A←AV[addr+X]	Nullseite indiziert
	(addr,X)	01 pp	2	6	X				X		A←AV[[addr+X]]	Vorindiziert indirekt
	(addr),Y	11 pp	2	5*	X				X		A←AV[addr+1, addr]+Y]	Nachindiziert indirekt
	addr16	0D ppqq	3	4	X				X		A←AV[addr16]	Erweitert direkt
	addr16,X od. Y	00011x01 ppqq	3	4*	X				X		A←AV[addr16+X] od. A←AV[addr16+Y]	Absolut indiziert

*Addiere eine Taktperiode, wenn eine Seiten-Grenze überschritten wird.
Im Objekt-Code zeigt "X" das Index-Register an: X=0 für Register Y, X=1 für Register X.

Tabelle 3-4. Eine Zusammenfassung des 6502-Befehlssatzes (Fortsetzung)

Typ	Befehl	Objekt-Code	Byte	Takt-Perioden	Status N	V	D	I	Z	C	Ausgeführte Operation	
Sekundäres Speicher-Ansprechen (Speicher-Operation) (Fortsetzung)	SBC										Subtrahiere Inhalt des Speicherplatzes, mit Borgen, vom Inhalt des Akkumulators	
	addr	E5 pp	2	3	X	X			X	X	A←A–[addr]–$\overline{C}$	Nullseite direkt
	addr,X	F5 pp	2	4	X	X			X	X	A←A–[addr+X]–$\overline{C}$	Nullseite indiziert
	(addr,X)	E1 pp	2	6	X	X			X	X	A←A–[[addr+X]]–$\overline{C}$	Vorindiziert indirekt
	(addr,)Y	F1 pp	2	5*	X	X			X	X	A←A–[[addr+1, addr]+Y]–$\overline{C}$	Nachindiziert indirekt
	addr16	ED ppqq	3	4	X	X			X	X	A←A–[addr16]–$\overline{C}$	Erweitert direkt
	addr16,X od. Y	11111x01 ppqq	3	4*	X	X			X	X	A←A–[addr16+X]–$\overline{C}$ od. A←A–[addr16+Y]–$\overline{C}$ (Beachte, daß der Übertrags-Wert das Komplement des Borgens ist.)	Absolut indiziert
	INC										Inkrementiere Inhalt des Speicherplatzes. Index nur über Register X.	
	addr	E6 pp	2	5	X				X		[addr]←[addr]+1	Nullseite direkt
	addr,X	F6 pp	2	6	X				X		[addr+X]←[addr+Y]+1	Nullseite indiziert
	addr16	EE ppqq	3	6	X				X		[addr16]←[addr16]+1	Erweitert direkt
	addr16,X	FE ppqq	3	7	X				X		[addr16+X]←[addr16+X]+1	Absolut indiziert
	DEC										Dekrementiere Inhalt des Speicherplatzes. Index nur über Register X.	
	addr	C6 pp	2	5	X				X		[addr]←[addr]–1	Nullseite direkt
	addr,X	D6 pp	2	6	X				X		[addr+X]←[addr+X]–1	Nullseite indiziert
	addr16	CE ppqq	3	6	X				X		[addr16]←[addr16]–1	Erweitert direkt
	addr16,X	DE ppqq	3	7	X				X		[addr16+X]–1	Absolut indiziert
	CPX										Vergleiche Inhalt von Register X mit jenem des Speicherplatzes. Nur die Statusflags werden beeinflußt.	
	addr	E4 pp	2	3	X				X	X	X–[addr]	Nullseite direkt
	addr16	EC ppqq	3	4	X				X	X	X–[addr16]	Erweitert direkt
	CPY										Vergleiche den Inhalt von Register Y mit jenem des Speicherplatzes. Nur die Statusflags werden beeinflußt	
	addr	C4 pp	2	3	X				X	X	Y–[addr]	Nullseite direkt
	addr16	CC ppqq	3	4	X				X	X	Y–[addr16]	Erweitert direkt

*Addiere eine Taktperiode, wenn eine Seiten-Grenze überschritten wird.
Im Objekt-Code zeigt "X" das Index-Register an: X=0 für Register Y, X=1 für Register X.

3-24

Tabelle 3-4. Eine Zusammenfassung des 6502-Befehlssatzes (Fortsetzung)

Typ	Befehl	Objekt-Code	Byte	Takt-Perioden	N	V	D	I	Z	C	Ausgeführte Operation
Sekundäres Speicher-Ansprechen (Speicher-Operation) (Fortsetzung)	ROL										Rotiere Inhalt des Speicherplatzes um ein Bit nach links durch den Übertrag. Index nur durch Register X.
	addr	26 pp	2	5	X				X	X	[addr]
	addr,X	36 pp	2	6	X				X	X	[addr+X]
	addr16	2E ppqq	3	6	X				X	X	[addr16]
	addr16,X	3E ppqq	3	7	X				X	X	[addr16+X]
	ROR										Rotiere Inhalt des Speicherplatzes um ein Bit nach rechts. Index nur durch Register X.
	addr	66 pp	2	5	X				X	X	[addr]
	addr,X	76 pp	2	6	X				X	X	[addr+X]
	addr16	6E pp	3	6	X				X	X	[addr16]
	addr16,X	7E ppqq	3	7	X				X	X	[addr16+X]
	ASL										Verschiebe Inhalt des Speicherplatzes arithmetisch nach links. Index nur durch Register X.
	addr	06 pp	2	5	X				X	X	[addr]
	addr,X	16 pp	2	6	X				X	X	[addr+X]
	addr16	0E ppqq	3	6	X				X	X	[addr16]
	addr16,X	1E ppqq	3	7	X				X	X	[addr16+X]

*Addiere eine Taktperiode, wenn eine Seiten-Grenze überschritten wird.
Im Objekt-Code zeigt "X" das Index-Register an: X=0 für Register Y, X=1 für Register X.

3-25

Tabelle 3-4. Eine Zusammenfassung des 6502-Befehlssatzes (Fortsetzung)

Typ	Befehl	Objekt-Code	Byte	Takt-Perioden	N	V	D	I	Z	C	Ausgeführte Operation
Sekundäres Speicher-Anspr. (Speicher-Oper.) (Forts.)	LSR										Verschiebe Inhalt des Speicherplatzeslogisch nach rechts. Index nur durch Register X.
	addr	46 pp	2	5	0				X	X	[addr]
	addr,X	56 pp	2	6	0				X	X	[addr+X]
	addr16	4E ppqq	3	6	0				X	X	[addr16]
	addr16,X	5E ppqq	3	7	0				X	X	[addr16,X]
Unmittelbar	LDA data	A9 pp	2	2	X				X		Lade Akkumulator mit unmittelbaren Daten A←data
	LDX data	A2 pp	2	2	X				X		Lade Index-Register X mit unmittelbaren Daten. X←data
	LDY data	A0 pp	2	2	X				X		Lade Index-Register Y mit unmittelbaren Daten. Y←data

*Addiere eine Taktperiode, wenn eine Seiten-Grenze überschritten wird.
Im Objekt-Code zeigt "X" das Index-Register an: X=0 für Register Y, X=1 für Register X.

Tabelle 3-4. Eine Zusammenfassung des 6502-Befehlssatzes (Fortsetzung)

Typ	Befehl	Objekt-Code	Byte	Takt-Perio-den	Status N	V	D	I	Z	C	Ausgeführte Operation
Unmittelbare Operation	ADC data	69 pp	2	2	X	X			X	X	Addiere unmittelbar mit Übertrag zum Akkumulator. Das Null-Flag ist in der Dezimalbetriebsart nicht gültig. A←A+data+C
	AND data	29 pp	2	2	X				X		UNDiere unmittelbar mit Akkumulator. A←A data
	CMP data	C9 pp	2	2	X				X	X	Vergleiche unmittelbar mit Akkumulator. Nur die Statusflags werden beeinflußt. A–data
	EOR data	49 pp	2	2	X				X		Exklusiv-ODERiere unmittelbar mit Akkumulator. A←A ⊻ data
	ORA data	09 pp	2	2	X				X		ODERiere unmittelbar mit Akkumulator. A←A V data
	SBC data	E9 pp	2	2	X	X			X	X	Subtrahiere unmittelbar, mit Borgen, vom Akkumulator. A←A–data–$\overline{C}$ (Beachten Sie, daß der Wert des Übertrags das Komplement des "Borgens" ist.)
	CPX data	E0 pp	2	2	X				X	X	Vergleiche unmittelbar mit Index-Register X. Nur die Statusflags werden beeinflußt. X–data
	CPY data	C0 pp	2	2	X				X	X	Vergleiche unmittelbar mit Index-Register Y. Nur die Statusflags werden beeinflußt. Y–data
Sprung	JMP label	4C ppqq	3	3							Springe zum nächsten Speicherplatz unter Verwendung erweiterter oder indirekter Adressierung. PC←label oder PC←[label]
	(label)	6C ppqq	3	5							

*Addiere eine Taktperiode, wenn eine Seiten-Grenze überschritten wird.
Im Objekt-Code zeigt "X" das Index-Register an: X=0 für Register Y, X=1 für Register X.

Tabelle 3-4. Eine Zusammenfassung des 6502-Befehlssatzes (Fortsetzung)

Typ	Befehl	Objekt-Code	Byte	Takt-Perio-den	Status N	V	D	I	Z	C	Ausgeführte Operation
Bedingte Verzweigung											Beachten Sie folgendes für alle bedingten Verzweigungs-Befehle: Wenn die Bedingung befriedigt wird, so wird die Versetzung zum Befehlszähler addiert, nachdem der Befehlszähler inkrementiert worden ist, um zu dem Befehl zu zeigen, der dem Verzweigungsbefehl folgt.
	BCC disp	90 pp	2	2**							Verzweige relativ, wenn Übertrags-Flag gelöscht ist. Wenn C=0, dann PC←PC+disp
	BCS disp	B0 pp	2	2**							Verzweige relativ, wenn Übertrags-Flag gesetzt ist. Wenn C=1, dann PC←PC+disp
	BEQ disp	F0 pp	2	2**							Verzweige relativ, wenn Ergebnis gleich null ist. Wenn Z=1, dann PC←PC+disp
	BMI disp	30 pp	2	2**							Verzweige relativ, wenn Ergebnis negativ ist. Wenn N=1, dann PC←PC+disp
	BNE disp	D0 pp	2	2**							Verzweige relativ, wenn Ergebnis nicht null ist. Wenn Z=0, dann PC←PC+disp
	BPL disp	10 pp	2	2**							Verzweige relativ, wenn Ergebnis positiv ist. Wenn N=0, dann PC←PC+disp
	BVC disp	50 pp	2	2**							Verzweige relativ, wenn Überlauf-Flag gelöscht ist. Wenn V=0, dann PC←PC+disp
	BVS disp	70 pp	2	2**							Verzweige relativ, wenn Überlauf-Flag gesetzt ist. Wenn V=1, dann PC←PC+disp

*Addiere eine Taktperiode, wenn die Verzweigung zu einem Speicherplatz auf derselben Seite auftritt. Addiere zwei Taktperioden, wenn die Verzweigung zu einer anderen Seite auftritt.

Tabelle 3-4. Eine Zusammenfassung des 6502-Befehlssatzes (Fortsetzung)

Typ	Befehl	Objekt-Code	Byte	Takt-Perioden	Status N	V	D	I	Z	C	Ausgeführte Operation
Unterprogramm-Aufruf und Rückkehr	JSR label	20 ppqq	3	6							Springe zu Unterprogramm, beginnend bei der Adresse, die durch die Bytes 2 und 3 des Befehls angegeben werden. Beachten Sie, daß der gespeicherte Befehlszähler zum letzten Byte des JSR-Befehls zeigt. [SP]←PC(HI) [SP−1]←PC(LO) SP←SP−2 PC←label
	RTS	60	1	6							Kehre von Unterprogramm zurück und inkrementiere hierbei den Befehlszähler, um zu dem Befehl nach dem JSR zu zeigen, der die Routine aufrief. PC(LO)←[SP+1] PC(HI)←[SP+2] SP←SP+2 PC←PC+1
Register-zu-Register-Operation	TAX	AA	1	2	X				X		Bringe Akkumulator-Inhalt zum Index-Register X. X←A
	TXA	8A	1	2	X				X		Bringe Inhalt des Index-Registers X zum Akkumulator A←X
	TAY	A8	1	2	X				X		Bringe Akkumulator-Inhalt zum Index-Register Y. Y←A
	TYA	98	1	2	X				X		Bringe Inhalt des Index-Registers Y zum Akkumulator. A←Y
	TSX	BA	1	2	X				X		Bringe Inhalt des Stapelzeigers zum Index-Register X. X←SP
	TXS	9A	1	2							Bringe Inhalt des Index-Registers X zum Stapelzeiger. SP←X

Tabelle 3-4. Eine Zusammenfassung des 6502-Befehlssatzes (Fortsetzung)

Typ	Befehl	Objekt-Code	Byte	Takt-Perioden	Status N	V	D	I	Z	C	Ausgeführte Operation
Register-Operation	DEX	CA	1	2	X				X		Dekrementiere Inhalt von Index-Register X. X←X−1
	DEY	88	1	2	X				X		Dekrementiere Inhalt von Index-Register Y. Y←Y−1
	INX	E8	1	2	X				X		Inkrementiere Inhalt von Index-Register X. X←X+1
	INY	C8	1	2	X				X		Inkrementiere Inhalt von Index-Register Y. Y←Y+1
	ROL A	2A	1	2	X				X	X	Rotiere Inhalt des Akkumulators nach links durch Übertrag.
	ROR A	6A	1	2	X				X	X	Rotiere Inhalt des Akkumulators nach rechts durch Übertrag.
	ASL A	0A	1	2	X				X	X	Verschiebe Inhalt des Akkumulators arithmetisch nach links.
	LSR A	4A	1	2	0				X	X	Verschiebe Inhalt des Akkumulators arithmetisch nach rechts.

Tabelle 3-4. Eine Zusammenfassung des 6502-Befehlssatzes (Fortsetzung)

Typ	Befehl	Objekt-Code	Byte	Takt-Perioden	Status						Ausgeführte Operation
					N	V	D	I	Z	C	
Stapel	PHA	48	1	3							Bringe Akkumulator-Inhalt zum Stapel. [SP]←A SP←SP−1
	PLA	68	1	4	X				X		Lade Akkumulator von der Spitze des Stapels ("Pull"). A←[SP+1] SP←SP+1
	PHP	08	1	3							Bringe Statusregister-Inhalt zum Stapel. [SP]←P SP←SP−1
	PLP	28	1	4	X	X	X	X	X	X	Lade Statusregister von der Spitze des Stapels ("Pull"). P←[SP+1] SP←SP+1
Unterbrechung	CLI	58	1	2				0			Gib Unterbrechungen durch Löschen des Unterbrechungs-Freigabe-Bits des Statusregisters frei. I←0
	SEI	78	1	2				1			Sperre Unterbrechungen. I←1
	RTI	40	1	6	X	X	X	X	X	X	Kehre von Unterbrechung zurück, speichere Status zurück. P←[SP+1] PC(LO)←[SP+2] PC(HI)←[SP+3] SP←SP+3 PC←PC+1
	BRK	00	1	7				1			Programmierte Unterbrechung. BRK kann nicht gesperrt werden. Der Programmzähler wird zweimal inkrementiert, bevor er im Stapel aufbewahrt wird. [SP]←PC(HI) [SP−1]←PC(LO) [SP−2]←P SP←SP−3 PC(HI)←[FFFF] PC(LO)←[FFFE] I←1 B←2

Tabelle 3-4. Eine Zusammenfassung des 6502-Befehlssatzes (Fortsetzung)

Typ	Befehl	Objekt-Code	Byte	Takt-Perioden	Status						Ausgeführte Operation
					N	V	D	I	Z	C	
Status	CLC	18	1	2						0	Lösche Übertrags-Flag C←0
	SEC	38	1	2						1	Setze Übertrags-Flag C←1
	CLD	D8	1	2			0				Lösche Dezimal-Betriebsart D←0
	SED	F8	1	2			1				Setze Dezimal-Betriebsart D←1
	CLV	B8	1	2		0					Lösche Überlauf-Flag V←0
	NOP	EA	1	2							Keine Operation

Tabelle 3-5. Tabelle der Objektcodes der Befehle des 6502 in numerischer Reihenfolge.

Objektcode	Befehl	
00	BRK	
01 pp	ORA	(addr,X)
05 pp	ORA	addr
06 pp	ASL	addr
08	PHP	
09 pp	ORA	data
0A	ASL	A
0D ppqq	ORA	addr16
0E ppqq	ASL	addr16
10 pp	BPL	disp
11 pp	ORA	(addr),Y
15 pp	ORA	addr,X
16 pp	ASL	addr,X
18	CLC	
19 ppqq	ORA	addr16,Y
1D ppqq	ORA	addr16,X
1E ppqq	ASL	addr16,X
20 ppqq	JSR	label
21 pp	AND	(addr,X)
24 pp	BIT	addr
25 pp	AND	addr
26 pp	ROL	addr
28	PLP	
29 pp	AND	data
2A	ROL	A
2C ppqq	BIT	addr16
2D ppqq	AND	addr16
2E ppqq	ROL	addr16
30 pp	BMI	disp
31 pp	AND	(addr),Y
35 pp	AND	addr,X
36 pp	ROL	addr,X
38	SEC	
39 ppqq	AND	addr16,Y
3D ppqq	AND	addr16,X
3E ppqq	ROL	addr16,X
40	RTI	
41 pp	EOR	(addr,X)
45 pp	EOR	addr
46 pp	LSR	addr
48	PHA	
49 pp	EOR	data
4A	LSR	A
4C ppqq	JMP	label
4D ppqq	EOR	addr16
4E ppqq	LSR	addr16
50 pp	BVC	disp
51 pp	EOR	(addr),Y
55 pp	EOR	addr,X
56 pp	LSR	addr,X
58	CLI	
59 ppqq	EOR	addr16,Y
5D ppqq	EOR	addr16,X
5E ppqq	LSR	addr16,X
60	RTS	
61 pp	ADC	(addr,X)
65 pp	ADC	addr
66 pp	ROR	addr
68	PLA	
69 pp	ADC	data
6A	ROR	A
6C ppqq	JMP	(label)
6D ppqq	ADC	addr16
6E ppqq	ROR	addr16
70 pp	BVS	disp
71 pp	ADC	(addr),Y
75 pp	ADC	addr,X
76 pp	ROR	addr,X
78	SEI	
79 ppqq	ADC	addr16,Y
7D ppqq	ADC	addr16,X
7E ppqq	ROR	addr16,X
81 pp	STA	(addr,X)
84 pp	STY	addr
85 pp	STA	addr
86 pp	STX	addr
88	DEY	
8A	TXA	
8C ppqq	STY	addr16
8D ppqq	STA	addr16
8E ppqq	STX	addr16
90 pp	BCC	disp
91 pp	STA	(addr),Y
94 pp	STY	addr,X
95 pp	STA	addr,X
96 pp	STX	addr,Y
98	TYA	
99 ppqq	STA	addr16,Y
9A	TXS	
9D ppqq	STA	addr16,X
A0 pp	LDY	data
A1 pp	LDA	(addr,X)
A2 pp	LDX	data
A4 pp	LDY	addr
A5 pp	LDA	addr
A6 pp	LDX	addr
A8	TAY	
A9 pp	LDA	data
AA	TAX	
AC ppqq	LDY	addr16
AD ppqq	LDA	addr16
AE ppqq	LDX	addr16
B0 pp	BCS	disp
B1 pp	LDA	(addr),Y
B4 pp	LDY	addr,X
B5 pp	LDA	addr,X
B6 pp	LDX	addr,Y
B8	CLV	
B9 ppqq	LDA	addr16,Y
BA	TSX	
BC ppqq	LDY	addr16,X
BD ppqq	LDA	addr16,X
BE ppqq	LDX	addr16,Y
C0 pp	CPY	data
C1 pp	CMP	(addr,X)
C4 pp	CPY	addr

Tabelle 3-5. Tabelle der Objektcodes der Befehle des 6502 in numerischer Reihenfolge (Fortsetzung)

Objektcode	Befehl	
C5 pp	CMP	addr
C6 pp	DEC	addr
C8	INY	
C9 pp	CMP	data
CA	DEX	
CC ppqq	CPY	addr16
CD ppqq	CMP	addr16
CE ppqq	DEC	addr16
D0 pp	BNE	disp
D1 pp	CMP	(addr),Y
D5 pp	CMP	addr,X
D6 pp	DEC	addr,X
D8	CLD	
D9 ppqq	CMP	addr16,Y
DD ppqq	CMP	addr16,X
DE ppqq	DEC	addr16,X
E0 pp	CPX	data
E1 pp	SBC	(addr,X)
E4 pp	CPX	addr
E5 pp	SBC	addr
E6 pp	INC	addr
E8	INX	
E9 pp	SBC	data
EA	NOP	
EC ppqq	CPX	addr16
ED ppqq	SBC	addr16
EE ppqq	INC	addr16
F0 pp	BEQ	disp
F1 pp	SBC	(addr),Y
F5 pp	SBC	addr,X
F6 pp	INC	addr,X
F8	SED	
F9 ppqq	SBC	addr16,Y
FD ppqq	SBC	addr16,X
FE ppqq	INC	addr16,X

Die folgenden Symbole werden in den Objektcodes in Tabelle 3-6 verwendet.

Auswahl der Adressier-Art:

aaa

000 vor-indiziert indirekt – (addr,X)
001 direkt – addr
010 unmittelbar – data
011 erweitert direkt – addr16
100 nach-indiziert indirekt – (addr),Y
101 Basis-Seite indiziert – addr,X
110 absolut indiziert – addr16,Y
111 absolut indiziert – addr16,X

bb

00 direkt – addr
01 erweitert direkt – addr16
10 Basis-Seite indiziert – addr,X
11 absolut indiziert – addr16,X

bbb

001 direkt – addr
010 Akkumulator – A
011 erweitert direkt – addr16
101 Basis-Seite indiziert – addr,X, addr,Y in STX
111 absolut indiziert – addr16,X, addr16,Y in STX

cc

00 unmittelbar – data
01 direkt – addr
11 erweitert direkt – addr16

ddd

000 unmittelbar – data
001 direkt – addr
011 erweitert direkt – addr16
101 Basis-Seite indiziert – addr,Y in LDX; addr,X in LDY
111 absolut indiziert – addr16,Y in LDX; addr16,X in LDY

pp Das zweite Byte eines Zwei- oder Drei-Byte-Befehls.

qq Das dritte Byte eines Drei-Byte-Befehls.

x Ein Bit, das die Adressen-Art wählt:
0 direkt – addr
1 erweitert direkt – addr16

Y Ein Bit, das die JMP-Adressier-Art wählt:
0 erweitert direkt – Markierung
1 indirekt – (Markierung)

Tabelle 3-6. Zusammenfassung der Objektcodes des 6502 mit Mnemoniks des 6800

Mnemonik	Operand	Objektcode	Bytes	Takt-Perioden	MC6800-Befehl
ADC		011aaa01			ADCA
	data	pp	2	2	data8
	addr	pp	2	3	addr8
	addr,X	pp	2	4	index
	(addr,X)	pp	2	6	
	(addr),Y	pp	2	5*	
	addr16	ppqq	3	4	addr16
	addr16,X	ppqq	3	4*	
	addr16,Y	ppqq	3	4*	
AND		001aaa01			ANDA
	data	pp	2	2	data8
	addr	pp	2	3	addr8
	addr,X	pp	2	4	index
	(addr,X)	pp	2	6	
	(addr),Y	pp	2	5*	
	addr16	ppqq	3	4	addr16
	addr16,X	ppqq	3	4*	
	addr16,Y	ppqq	3	4*	
ASL	A	000bbb10	1	2	ASLA
	addr	pp	1	5	
	addr,X	pp	2	6	ASL index
	addr16	ppqq	3	6	ASL addr16
	addr16,X	ppqq	3	7	
BCC	disp	90 pp	2	2**	BCC disp
BCS	disp	B0 pp	2	2**	BCS disp
BEQ	disp	F0 pp	2	2**	BEQ disp
BIT		0010x100			BITA
	addr	pp	2	3	addr8
	addr16	ppqq	3	4	addr16
BMI	disp	30 pp	2	2**	BMI disp
BNE	disp	D0 pp	2	2**	BNE disp
BPL	disp	10 pp	2	2**	BPL disp
BRK		00	1	7	(SWI)
BVC	disp	50 pp	2	2**	BVC disp
BVS	disp	70 pp	2	2**	BVS disp
CLC		18	1	2	CLC
CLD		D8	1	2	
CLI		58	1	2	CLI
CLV		B8	1	2	CLV

Addiere eine Taktperiode, wenn eine Seiten-Grenze überschritten wird.
**Addiere eine Taktperiode, wenn Verzweigung zum Speicherplatz in derselben Seite auftritt; addiere zwei Taktperioden, wenn Verzweigung zu anderer Seite auftritt.

Tabelle 3-6. Zusammenfassung der Objektcodes des 6502 mit Mnemoniks des 6800 (Fortsetzung)

Mnemonik	Operand	Objektcode	Bytes	Takt-Perioden	MC6800-Befehl
CMP		110aaa01			CMPA
	data	pp	2	2	data8
	addr	pp	2	3	addr8
	addr,X	pp	2	4	index
	(addr,X)	pp	2	6	
	(addr),Y	pp	2	5*	
	addr16	ppqq	3	4	addr16
	addr16,X	ppqq	3	4*	
	addr16,Y	ppqq	3	4*	
CPX		1110cc00			CPX
	data	pp	2	2	data8
	addr	pp	2	3	addr8
	addr16	ppqq	3	4	addr16
CPY		1100cc00			
	data	pp	2	2	
	addr	pp	2	3	
	addr16	ppqq	3	4	
DEC		110bb110			DEC
	addr	pp	2	5	
	addr,X	pp	2	6	index
	addr16	ppqq	3	6	addr16
	addr16,X	ppqq	3	7	
DEX		CA	1	2	DEX
DEY		88	1	2	
EOR		010aaa01			EORA
	data	pp	2	2	data8
	addr	pp	2	3	addr8
	addr,X	pp	2	4	index
	(addr,X)	pp	2	6	
	(addr),Y	pp	2	5*	
	addr16	ppqq	3	4	addr16
	addr16,X	ppqq	3	4*	
	addr16,Y	ppqq	3	4*	
INC		111bb110			INC
	addr	pp	2	5	
	addr,X	pp	2	6	index
	addr16	ppqq	3	6	addr16
	addr16,X	ppqq	3	7	
INX		E8	1	2	INX
INY		C8	1	2	
JMP		01y01100			JMP
	label	ppqq	3	3	addr16
	(label)	ppqq	3	5	
JSR	label	20 ppqq	3	6	JSR addr16

Addiere eine Taktperiode, wenn eine Seiten-Grenze überschritten wird.
**Addiere eine Taktperiode, wenn Verzweigung zum Speicherplatz in derselben Seite auftritt; addiere zwei Taktperioden, wenn Verzweigung zu anderer Seite auftritt.

Tabelle 3-6. Zusammenfassung der Objektcodes des 6502 mit Mnemoniks des 6800 (Fortsetzung)

Mnemonik	Operand	Objektcode	Bytes	Takt-Perioden	MCS6800-Befehl
LDA		101aaa01			LDAA
	data	pp	2	2	data8
	addr	pp	2	3	addr8
	addr,X	pp	2	4	index
	(addr,X)	pp	2	6	
	(addr),Y	pp	2	5*	
	addr16	ppqq	3	4	addr16
	addr16,X	ppqq	3	4*	
	addr16,Y	ppqq	3	4*	
LDX		101ddd10			LDX
	data	pp	2	2	(data8)
	addr	pp	2	3	addr8
	addr,Y	pp	2	4	(index)
	addr16	ppqq	3	4	addr16
	addr16,Y	ppqq	3	4*	
LDY		101ddd00			
	data	pp	2	2	
	addr	pp	2	3	
	addr,X	pp	2	4	
	addr16	ppqq	3	4	
	addr16,X	ppqq	3	4*	
LSR	A	010bbb10	1	2	LSRA
	addr	pp	2	5	
	addr,X	pp	2	6	LSR index
	addr16	ppqq	3	6	LSR addr16
	addr16,X	ppqq	3	7	
NOP		EA	1	2	NOP
ORA		000aaa01			ORAA
	data	pp	2	2	data8
	addr	pp	2	3	addr8
	addr,X	pp	2	4	index
	(addr,X)	pp	2	6	
	(addr),Y	pp	2	5*	
	addr16	ppqq	3	4	addr16
	addr16,X	ppqq	3	4*	
	addr16,Y	ppqq	3	4*	
PHA		48	1	3	PSHA
PHP		08	1	3	
PLA		68	1	4	PULA
PLP		28	1	4	
ROL	A	001bbb10	1	2	ROLA
	addr	pp	2	5	
	addr,X	pp	2	6	ROL index
	addr16	ppqq	3	6	ROL addr16
	addr16,X	ppqq	3	7	

Addiere eine Taktperiode, wenn eine Seiten-Grenze überschritten wird.
**Addiere eine Taktperiode, wenn Verzweigung zum Speicherplatz in derselben Seite auftritt; addiere zwei Taktperioden, wenn Verzweigung zu anderer Seite auftritt.

Tabelle 6. Zusammenfassung der Objektcodes des 6502 mit Mnemoniks des 6800 (Fortsetzung)

Mnemonik	Operand	Objektcode	Bytes	Takt-Perioden	MC6800-Befehl
ROR	A	011bbb10	1	2	RORA
	addr	pp	2	5	
	addr,X	pp	2	6	ROR index
	addr16	ppqq	3	6	ROR addr16
	addr16,X	ppqq	3	7	
RTI		40	1	6	RTI
RTS		60	1	6	RTS
SBC		111aaa01			SBCA
	data	pp	2	2	data8
	addr	pp	2	3	addr8
	addr,X	pp	2	4	index
	(addr,X)	pp	2	6	
	(addr),Y	pp	2	5*	
	addr16	ppqq	3	4	addr16
	addr16,X	ppqq	3	4*	
	addr16,Y	ppqq	3	4*	
SEC		38	1	2	SEC
SED		F8	1	2	
SEI		78	1	2	SEI
STA		100aaa01			STAA
	addr	pp	2	3	addr8
	addr,X	pp	2	4	index
	(addr,X)	pp	2	6	
	(addr),Y	pp	2	6	
	addr16	ppqq	3	4	addr16
	addr16,X	ppqq	3	5	
	addr16,Y	ppqq	3	5	
STX		100bb110			STX
	addr	pp	2	3	addr8
	addr,Y	pp	2	4	(index)
	addr16	ppqq	3	4	addr16
STY		100bb100			
	addr	pp	2	3	
	addr,X	pp	2	4	
	addr16	ppqq	3	4	
TAX		AA	1	2	
TAY		A8	1	2	
TSX		BA	1	2	TSX
TXA		8A	1	2	
TXS		9A	1	2	TXS
TYA		98	1	2	

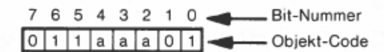
Addiere eine Taktperiode, wenn eine Seiten-Grenze überschritten wird.
**Addiere eine Taktperiode, wenn Verzweigung zum Speicherplatz in derselben Seite auftritt; addiere zwei Taktperioden, wenn Verzweigung zu anderer Seite auftritt.

ADC – ADD-MEMORY, WITH CARRY, TO ACCUMULATOR (ADDIERE SPEICHER, MIT ÜBERTRAG, ZUM AKKUMULATOR)

Dieser Befehl verwendet acht Arten der Adressierung des Datenspeichers und gestattet die Addition des Inhalts des Datenspeichers und des Übertrags-Status zum Akkumulator.Die acht Arten der Speicher-Adressierung sind:

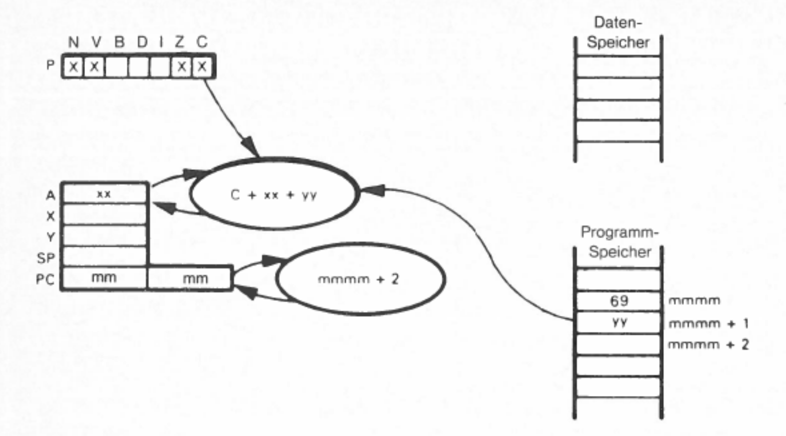
1) Unmittelbar – ADC data
2) Absolut (direkt) – ADC addr16
3) Null-Seite (direkt) – ADC addr
4) Vor-indiziert mit Indexregister X – ADC (addr,X)
5) Nach-indiziert mit Indexregister Y – ADC (addr),Y
6) Null-Seite indiziert mit Indexregister X – ADC addr,X
7) Absolut indiziert mit Indexregister X – ADC addr16,X
8) Absolut indiziert mit Indexregister Y – ADC addr16,Y

Das erste Byte des Objektcodes bestimmt wie folgt, welche Adressier-Art ausgewählt wird:



Bit-Wert für aaa	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
000	61	Indirekt, vor-indiziert mit X	2
001	65	Null-Seite (direkt)	2
010	69	Unmittelbar	2
011	6D	Absolut (direkt)	3
100	71	Indirekt, nach-indiziert mit Y	2
101	75	Null-Seite indiziert mit X	2
110	79	Absolut indiziert mit Y	3
111	7D	Absolut indiziert mit X	3

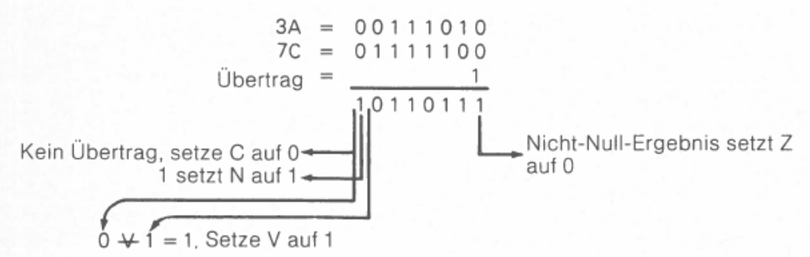
Wir können den ADC-Befehl mit unmittelbarer Adressierung wie anschließend gezeigt illustrieren. Für andere Adressier-Arten schlagen Sie entweder die Besprechung der Adressier-Arten oder die Beschreibung anderer arithmetischer oder logischer Befehle nach, da die übrigen Darstellungen unterschiedliche Adressier-Arten zeigen.



Addiere den Inhalt des nächsten Programmspeicher-Bytes (Adressier-Art, gewählt durch die Bits 2, 3 und 4 des Bytes im Befehlsregister) mit dem Übertrags-Status zum Akkumulator. Es sei angenommen $xx = 3A_{16}$, $yy = 7C_{16}$, C = 1. Nachdem der Befehl

ADC #$7C

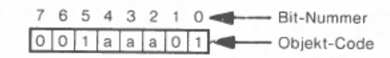
ausgeführt wurde, wird der Akkumulator $B7_{16}$ enthalten.



ADC ist der einzige Additionsbefehl des 6502. Um ihn in Einzelbyte-Operationen oder zur Addition der niederwertigen Bytes zweier Multi-Bytezahlen zu verwenden, muß ein vorhergehender Befehl ausdrücklich den Übertrag auf Null setzen, so daß er die Operation nicht beeinflußt. Beachten Sie, daß der Mikroprozessor 6502 keinen Additionsbefehl besitzt, der den Übertrag nicht beinhaltet. ADC wird entweder binäre oder dezimale (BCD-) Addition ausführen, abhängig davon, ob das Flag für den Dezimal-Betrieb 0 oder 1 ist.

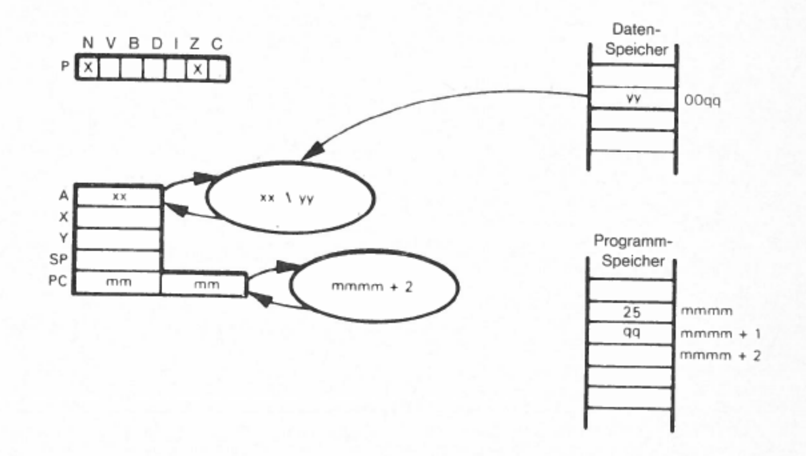
AND – AND MEMORY WITH ACCUMULATOR (UNDiere SPEICHER MIT AKKUMULATOR)

Dieser Befehl UNDiert logisch den Inhalt eines Speicherplatzes mit dem Inhalt des Akkumulators. Dieser Befehl bietet die gleichen Speicher-Adressiermöglichkeiten wie der ADC-Befehl. Das erste Byte des Objektcodes wählt die Adressier-Art wie folgt:



Bit-Wert für aaa	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
000	21	Indirekt, vor-indiziert mit X	2
001	25	Null-Seite (direkt)	2
010	29	Unmittelbar	2
011	2D	Absolut (direkt)	3
100	31	Indirekt, nach-indiziert mit Y	2
101	35	Null-Seite indiziert mit X	2
110	39	Absolut indiziert mit Y	3
111	3D	Absolut indiziert mit X	3

Wir wollen den AND-Befehl mit einer (direkten) Adressierung der Seite Null illustrieren. Sehen Sie sich die Besprechung der Adressier-Methoden und anderer arithmetischer und logischer Befehle für Beispiele der anderen Adressier-Arten an.



UNDiere logisch den Inhalt des gewählten Speicherbytes mit dem Akkumulator und speichere das Ergebnis in den Akkumulator. Es sei angenommen xx = FC_{16} und yy = 13_{16}. Nach dem Befehl

AND $40

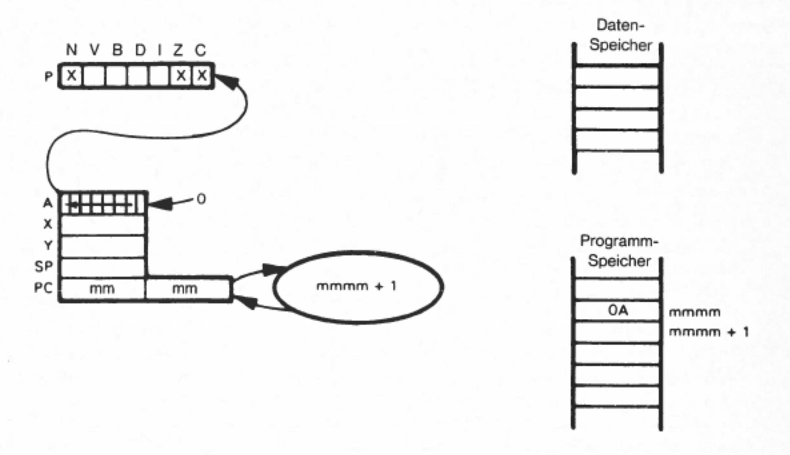
(angenommen, daß yy in Speicherplatz 0040 liegt) wird der Akkumulator 10_{16} enthalten:

```
            FC = 11111100
            13 = 00010011
                 --------
                 00010000
                 |      |
0 in Bit 7 setzt N auf 0 <-+      +-> Nicht-Null-Ergebnis setzt Z auf 0
```

AND ist ein häufig verwendeter logischer Befehl.

ASL – SHIFT ACCUMULATOR OR MEMORY BYTE LEFT (SCHIEBE AKKUMULATOR ODER SPEICHERBYTE NACH LINKS)

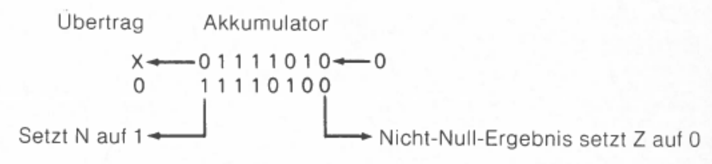
Dieser Befehl führt eine arithmetische Links-Verschiebung um ein Bit des Inhalts des Akkumulators oder des Inhalts des gewählten Speicherbytes aus. Betrachten Sie zuerst die Verschiebung des Akkumulators:



Der Akkumulator enthalte angenommen $7A_{16}$. Bei der Ausführung des Befehls

ASL A

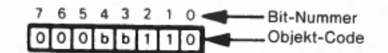
wird der Übertrags-Status auf 0 gesetzt, das Negativ-Bit auf 1, der Null-Status auf 0 und es wird $F4_{16}$ in den Akkumulator gespeichert.



Der ASL-Befehl verwendet vier Optionen für die Adressierung des Datenspeichers:

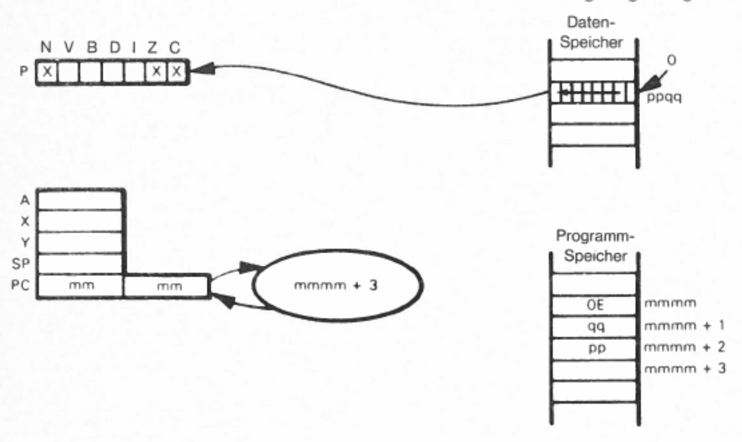
1) Null-Seite (direkt) – ASL addr
2) Absolut (direkt) – ASL addr16
3) Null-Seite indiziert mit Indexregister X – ASL addr,X
4) Absolut indiziert mit Indexregister X – ASL addr16,X

Das erste Byte des Objektcodes bestimmt, welche Adressier-Art wie folgt gewählt wurde:



Bit-Wert für bb	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
00	06	Null-Seite (direkt)	2
01	0E	Absolut (direkt)	3
10	16	Null-Seite indiziert mit X	2
1	1E	Absolut indiziert mit X	3

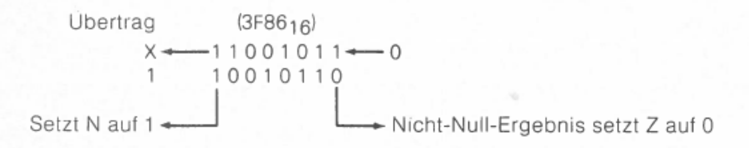
Wir wollen den ASL-Befehl mit absoluter (direkter) Adressierung zeigen. Die anderen Adressier-Arten sind bei anderen Befehls-Beschreibungen gezeigt.



Es sei angenommen ppqq = $3F86_{16}$ und der Inhalt von ppqq sei CB_{16}. Nach Ausführung eines Befehls

ASL $3F86

wird der Inhalt des Speicherplatzes $3F86_{16}$ auf 96_{16} geändert und der Übertrag auf 1 gesetzt:



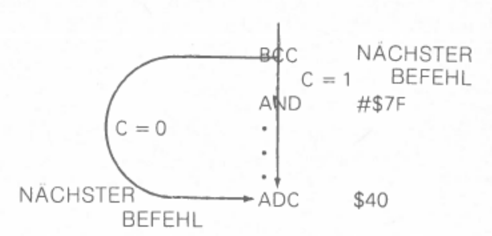
Der ASL-Befehl wird häufig in Manipulations-Routinen und als Standard-Logikbefehl verwendet. Beachten Sie, daß ein einzelner ASL-Befehl seinen Operanden mit 2 multipliziert.

BCC – BRANCH IF CARRY CLEAR (C = 0) (VERZWEIGE, WENN ÜBERTRAG GELÖSCHT)

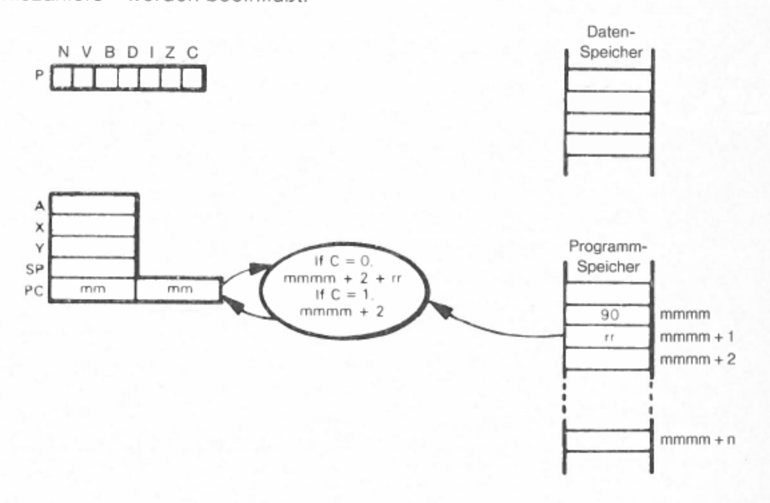
Dieser Befehl ist eine Verzweigung mit relativer Adressierung, bei dem die Verzweigung nur ausgeführt wird, wenn der Übertrags-Status gleich 0 ist. Andernfalls wird der nächste Befehl ausgeführt.

BCC
90

In der folgenden Befehls-Sequenz:



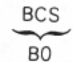
wird der Befehl ADC $40 unmittelbar nach dem BCC-Befehl ausgeführt, wenn der Übertrags-Status gleich 0 ist. Der Befehl AND #$7F wird ausgeführt, wenn der Übertrags-Status gleich 1 ist. Die relative Adressierung arbeitet wie in der nächsten Abbildung gezeigt und wie in der Besprechung der früher dargelegten Adressier-Methoden. Keine Status und keine Register – mit Ausnahme des Befehlszählers – werden beeinflußt.



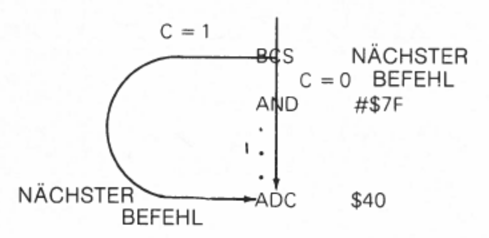
Wenn der Übertrag Null ist, so addiert dieser Befehl den Inhalt des zweiten Objektcode-Bytes (genommen als 8-Bit-Versetzung mit Vorzeichen) zum Inhalt des Befehlszählers plus 2. Dies wird die Speicher-Adresse für den nächsten auszuführenden Befehl. Der vorhergehende Inhalt des Befehlszählers geht verloren.

BCS – BRANCH IF CARRY SET (C = 1) (VERZWEIGE, WENN ÜBERTRAG GESETZT)

Dieser Befehl arbeitet wie der BCC-Befehl mit der Ausnahme, daß die Verzweigung nur ausgeführt wird, wenn der Übertrags-Status gleich 1 ist. Andernfalls wird der nächste Befehl ausgeführt.



In der folgenden Befehls-Sequenz:



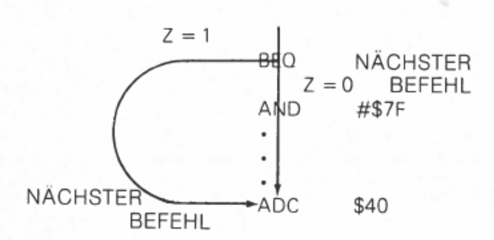
wird der Befehl ADC $40 direkt nach dem BCS-Befehl ausgeführt, wenn der Übertrags-Status gleich 1 ist. Der Befehl AND #$7F wird ausgeführt, wenn der Übertrags-Status gleich 0 ist.

BEQ – BRANCH IF EQUAL TO ZERO (Z = 1) (VERZWEIGE, WENN GLEICH NULL)

Dieser Befehl ähnelt weitgehend dem BCC-Befehl mit der Ausnahme, daß die Verzweigung ausgeführt wird, wenn der Null-Status gleich 1 ist. Andernfalls wird der nächste Befehl ausgeführt.

BEQ
F0

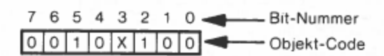
In der folgenden Sequenz:



wird der Befehl ADC $40 direkt nach dem BEQ-Befehl ausgeführt, wenn der Null-Status gleich 1 ist. Der Befehl AND #$7F wird ausgeführt, wenn der Null-Status gleich 0 ist.

BIT – BIT-TEST (BIT-TEST)

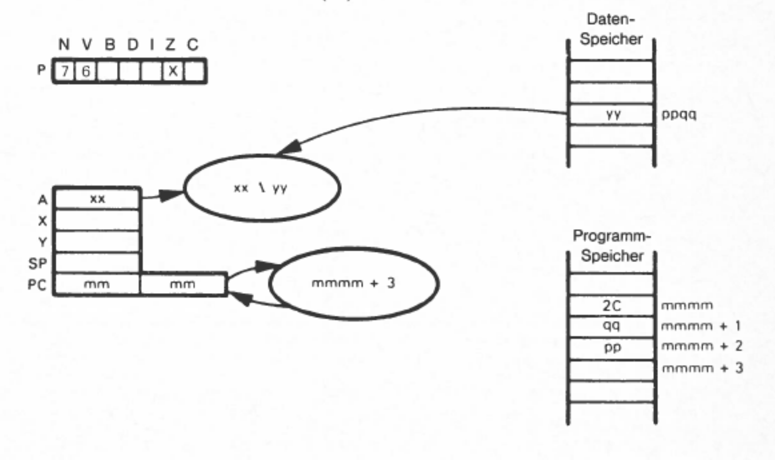
Dieser Befehl UNDiert logisch den Inhalt des Akkumulators mit dem Inhalt eines gewählten Speicherplatzes, setzt die Bedingungs-Flags entsprechend, ändert jedoch nicht den Inhalt des Akkumulators oder des Speicherbytes. Die einzigen zulässigen Adressier-Arten sind absolut (direkt) und Null-Seite (direkt). Das erste Byte des Objektcodes bestimmt die Adressier-Art wie folgt:



Bit-Wert für aaa	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
0	24	Null-Seite (direkt)	2
1	2C	Absolut (direkt)	3

Wir wollen den BIT-Befehl unter Verwendung der absoluten (direkten) Adressierung illustrieren. Für den Nullseiten-Betrieb siehe den AND-Befehl und die Besprechung der Adressier-Arten. Wir sollten beachten, daß Bit einen ziemlich ungewöhnlichen Einfluß auf die Status-Flags besitzt, da es das Z-Flag entsprechend dem Ergebnis der logischen UND-Operation setzt, jedoch die N- und V-Flags entsprechend den Bits 7 und 6 des Inhalts des zu testenden Speicherplatzes setzt. Das heißt:

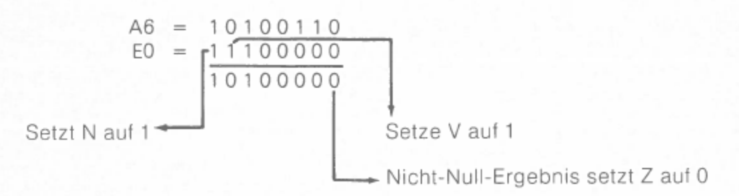
$$Z = 1 \text{ wenn } A \wedge (M) = 0,\; Z = 0 \text{ wenn } A \wedge (M) \neq 0$$
$$N = \text{Bit 7 von } (M)$$
$$V = \text{Bit 6 von } (M)$$



UNDiere logisch den Inhalt des Akkumulators mit dem Inhalt des spezifizierten Speicherplatzes und setze das Null-Flag entsprechend. Setze das Negativ- und Uberlauf-Flag entsprechend den Bits 7 und 6 des gewählten Speicherplatzes. Es sei angenommen xx = $A6_{16}$, yy = $E0_{16}$ und ppqq = 1641_{16}. Nachdem der Befehl

BIT $1641

ausgeführt wurde, wird der Akkumulator noch $A6_{16}$ und der Speicherplatz 1641_{16} noch $E0_{16}$ enthalten, die Status werden jedoch wie folgt modifiziert worden sein:

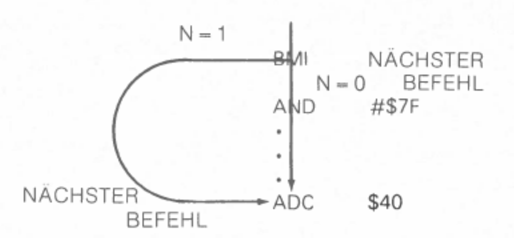


BIT-Befehle gehen häufig bedingten Verzweigungsbefehlen voraus. BIT-Befehle werden auch zur Ausführung von Maskier-Funktionen an Daten eingesetzt.

BMI – BRANCH IF MINUS (N = 1) (VERZWEIGE, WENN MINUS)



Dieser Befehl arbeitet wie der BCC-Befehl mit der Ausnahme, daß die Verzweigung nur ausgeführt wird, wenn das Negativ-Flag gleich 1 ist. Andernfalls wird der nächste Befehl ausgeführt.
In der folgenden Befehls-Sequenz

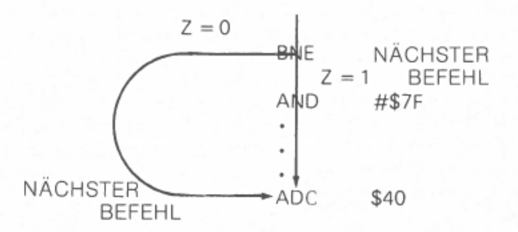


wird der Befehl ADC $40 direkt nach dem BMI-Befehl ausgeführt, wenn das Negativ-Flag gleich 1 ist. Der Befehl AND #$7F wird ausgeführt, wenn das Negativ-Flag 0 ist.

BNE – BRANCH IF NOT EQUAL TO ZERO (Z = 0) (VERZWEIGE, WENN NICHT GLEICH NULL)

BNE
D0

Dieser Befehl ist identisch mit dem BCD-Befehl mit der Ausnahme, daß die Verzweigung nur ausgeführt wird, wenn der Null-Status gleich 0 ist. Andernfalls wird der nächste Befehl in der Sequenz ausgeführt.
In der folgenden Befehls-Sequenz

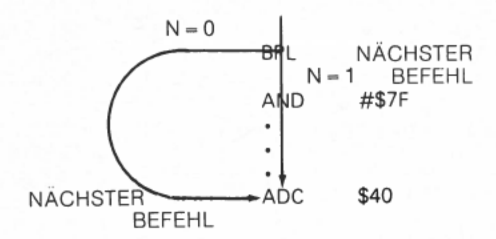


wird der Befehl ADC $40 direkt nach dem BNE-Befehl ausgeführt, wenn der Null-Status gleich 1 ist. Der Befehl AND #$7F wird ausgeführt, wenn der Null-Status 1 ist.

BPL – BRANCH IF PLUS (N = 0) (VERZWEIGE, WENN PLUS)



Dieser Befehl arbeitet wie der BCC-Befehl mit der Ausnahme, daß die Verzweigung nur ausgeführt wird, wenn der Negativ-Status 0 ist. Andernfalls wird der nächste Befehl in der Sequenz ausgeführt.
In der folgenden Befehls-Sequenz



wird der Befehl ADC $40 direkt nach dem BPL-Befehl ausgeführt, wenn das Negativ-Flag 0 ist. Der Befehl AND #$7F wird ausgeführt, wenn das Negativ-Flag 1 ist.

BRK – FORCE BREAK (TRAP OR SOFTWARE INTERRUPT) (ERZWINGE UNTERBRECHUNG – FALLE ODER SOFTWARE-UNTERBRECHUNG –)



Der Befehlszähler wird um Zwei inkrementiert, und der Break-Status wird auf 1 gesetzt, dann wird der Befehlszähler und das Status- (P-) Register auf den Stapel gebracht (pushed). Die Register und die entsprechenden Speicherplätze, in die diese gebracht werden, sind folgende:

Speicherplatz	Register
(Stapelzeiger enthält ss am Beginn der Befehlsausführung)	
01ss	Hochwertiges Byte des Befehlszählers
01ss – 1	Niederwertiges Byte des Befehlszählers
01ss – 2	Status- (P-) Register mit B = 1
(Stapelzeiger enthält ss – 3 am Ende der Befehlsausführung)	

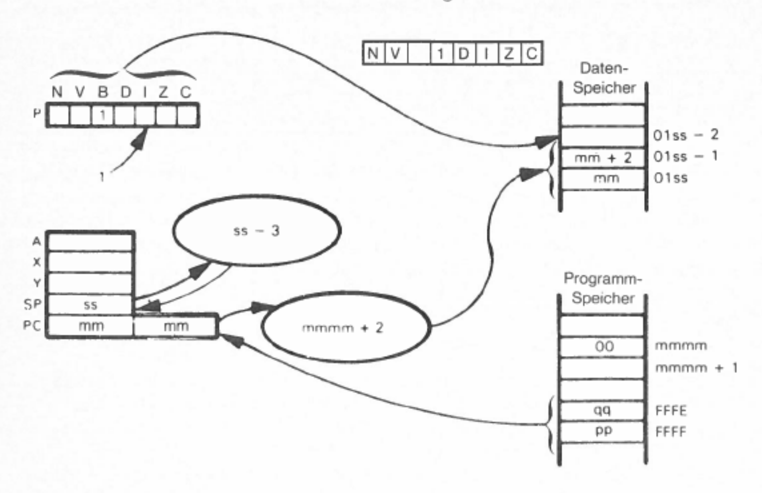
Das Unterbrechungsmasken-Bit wird dann auf 1 gesetzt. Dies sperrt die Möglichkeiten des 6502, Unterbrechungen zu bedienen, das heißt, der Prozessor wird nicht auf eine Unterbrechung von einem peripheren Baustein reagieren. Der Inhalt des Unterbrechungs-Zeigers (Speicher-Adressen $FFFE_{16}$ und $FFFF_{16}$) wird dann in den Befehlszähler geladen.

Der BRK-Befehl kann für eine Vielzahl von Funktionen verwendet werden. Er liefert die Möglichkeit, einen Haltepunkt für Fehlersuch-Zwecke zu setzen oder er kann die Steuerung zu einem besonders wichtigen Software-System, wie einem Disketten-Betriebssystem oder einem Monitor, transferieren. Beachten Sie, daß der Programmierer den erforderlichen Code einsetzen muß, um einen BRK-Befehl von einer regulären Unterbrechungs-Antwort zu instruieren. Die Codierung hierfür prüft den Wert des B-Statusflags im Stapel wie folgt:

```
PLA              ;HOLE STATUSREGISTER
PHA              ;ABER LASSE ES AUCH AUF DEM STAPEL
AND    #$10      ;IST DER BREAK-STATUS GESETZT?
BNE    BRKP      ;JA, FUHRE UNTERBRECHUNG AUS
```

Beachten Sie, daß der Operationscode für BRK gleich 00 ist. Diese Auswahl des Operationscodes bedeutet, daß BRK zum Einbringen von Programmen in PROMs mit Schmelzsicherungen verwendet werden kann, da das Durchschmelzen aller Brücken den Inhalt des Wortes auf 00 bringt. Daher kann ein fehlerhafter Befehl durch Änderung des ersten Objektcode-Bytes auf 00 korrigiert werden, wodurch ein Muster über die Unterbrechungsvektor-Routine eingesetzt wird. Erinnern sie sich daran, daß ein Bit in einem PROM mit Schmelzbrücken auf 0 gesetzt werden kann (indem die Schmelzbrücke durchgeschmolzen wird), jedoch nicht auf 1 zurückgesetzt werden kann, wenn die Brücke durchgeschmolzen worden ist. Derartige PROMs sind nicht löschbar.

Die Arbeitsweise des BRK-Befehls kann wie folgt illustriert werden:

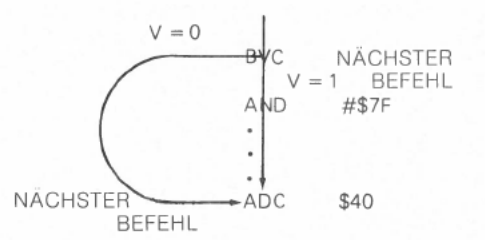


Der abschließende Inhalt des Befehlszählers ist ppqq, wobei pp den Inhalt des Speicherplatzes $FFFF_{16}$ darstellt und qq den Inhalt des Speicherplatzes $FFFE_{16}$. Beachten Sie, daß der Stapel immer auf Seite 1 des Speichers liegt, d.h. die acht höchstwertigen Bits der Stapel-Adresse sind immer 01_{16}.

BVC – BRANCH IF OVERFLOW CLEAR (V = 0) (VERZWEIGE, WENN ÜBERLAUF GELÖSCHT)

BVC
50

Dieser Befehl arbeitet wie der BCC-Befehl mit der Ausnahme, daß die Verzweigung nur ausgeführt wird, wenn der Überlauf-Status 0 ist. Andernfalls wird der nächste Befehl in der Sequenz ausgeführt.
In der folgenden Befehls-Sequenz



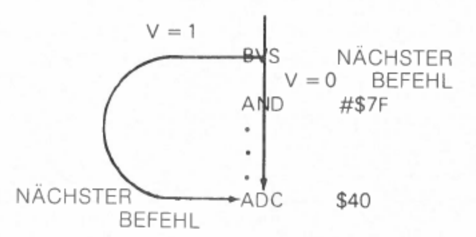
wird der Befehl ADC $40 direkt nach dem BVC-Befehl ausgeführt, wenn der Überlauf-Status 0 ist. Der Befehl AND #$7F wird ausgeführt, wenn der Überlauf-Status gleich 1 ist.

BVS – BRANCH IF OVERFLOW SET (V = 1) (VERZWEIGE, WENN ÜBERLAUF GESETZT)

BVS

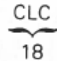
70

Dieser Befehl arbeitet wie der BCC-Befehl mit der Ausnahme, daß die Verzweigung nur ausgeführt wird, wenn der Überlauf-Status gleich 1 ist. Andernfalls wird der nächste Befehl in der Sequenz ausgeführt.
In der folgenden Befehls-Sequenz

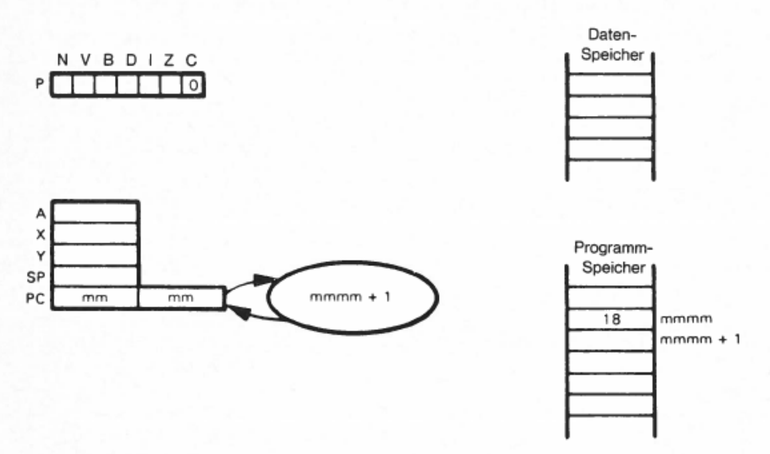


wird der Befehl ADC $40 direkt nach dem BVS-Befehl ausgeführt, wenn der Überlauf-Status gleich 1 ist. Der Befehl AND #$7F wird ausgeführt, wenn der Überlauf-Status gleich 0 ist.

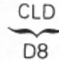
CLC – CLEAR CARRY (LÖSCHE ÜBERTRAG)



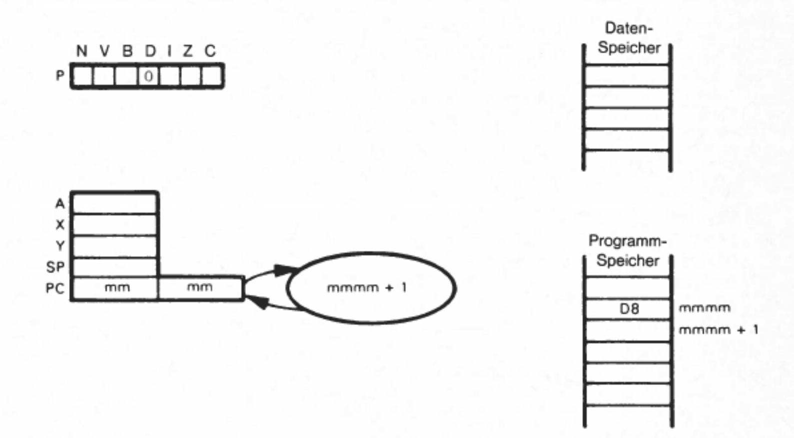
Löscht den Übertrags-Status. Kein anderer Status oder Register-Inhalt wird beeinflußt. Beachten Sie, daß dieser Befehl als Teil der normalen Additions-Operation erforderlich ist, da der einzige beim Mikroprozessor 6502 verfügbare Additionsbefehl ADC ist, der auch den Übertrags-Status addiert. Dieser Befehl ist also beim Beginn einer Multi-Byte-Addition erforderlich, da es niemals einen Übertrag in das niedrigstwertige Byte gibt.



CLD – CLEAR DECIMAL MODE (LÖSCHE DEZIMAL-BETRIEBSART)



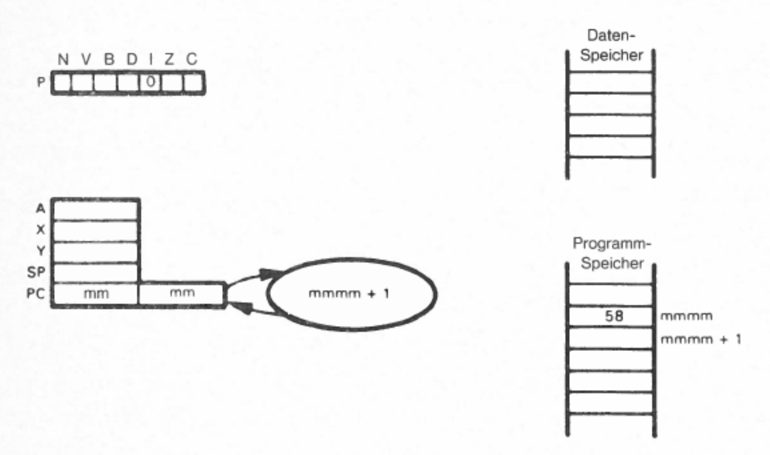
Löscht den Dezimal-Betriebsart-Status. Kein anderer Status oder Register-Inhalt wird beeinflußt. Dieser Befehl wird zum Zurückstellen des 6502-Prozessors in die binäre Betriebsart verwendet, bei der ADC- und SBC-Befehle binäre anstatt BCD-Ergebnisse erzeugen. Dieser Befehl kann verwendet werden um zu sichern, daß die Betriebsart in Situationen binär ist, in denen es unsicher sein kann, ob der Dezimalbetriebs-Status kürzlich gesetzt oder gelöscht wurde.



CLI – CLEAR INTERRUPT MASK (ENABLE INTERRUPTS) (LÖSCHE UNTERBRECHUNGS-MASKE – GIB UNTERBRECHUNGEN FREI –)



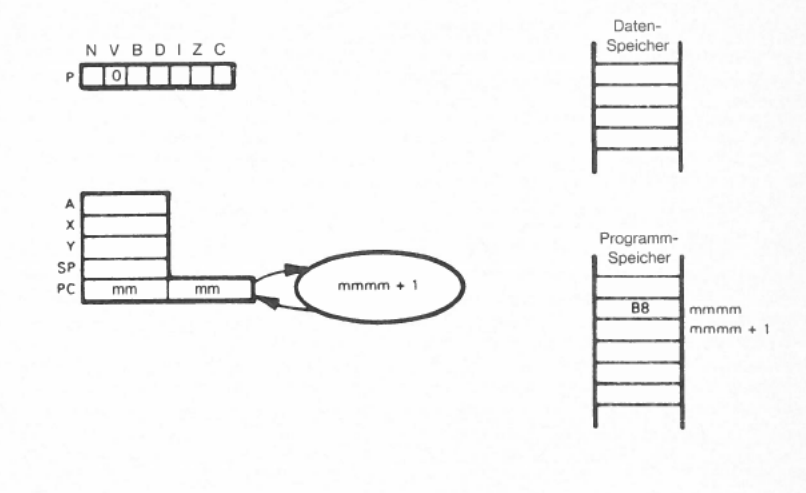
Löscht das Unterbrechungs-Maskenbit im Status- (P-) Register. Dieser Befehl gibt die Unterbrechungs-Möglichkeit des 6502 frei, d.h., der 6502 wird auf die Unterbrechungs-Anforderungs-Steuerleitung reagieren. Es werden keine weiteren Register oder Status beeinflußt. Beachten Sie, daß das I-Bit eine Maske oder ein Sperr-Bit ist. Es muß gelöscht werden, um Unterbrechungen freizugeben, und gesetzt werden, um sie zu sperren.



CLV – CLEAR OVERFLOW (LÖSCHE ÜBERLAUF)

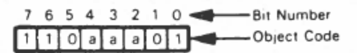


Löscht das Überlauf-Bit im Status-Register. Keine anderen Register oder Status werden beeinflußt. Beachten Sie, daß der 6502 keinen Befehl für SET OVERFLOW (Setze Überlauf) besitzt.



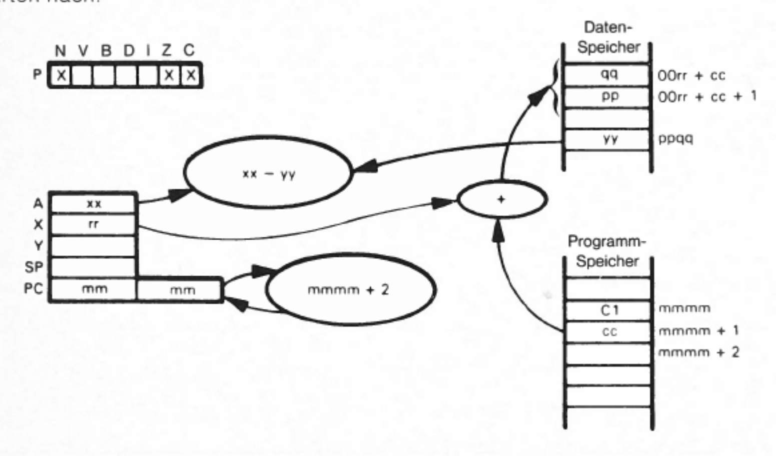
CMP – COMPARE MEMORY WITH ACCUMULATOR (VERGLEICHE SPEICHER MIT AKKUMULATOR)

Dieser Befehl subtrahiert den Inhalt des gewählten Speicher-Bytes vom Akkumulator, setzt die Bedingungs-Flags entsprechend, beeinflußt jedoch nicht den Inhalt des Akkumulators oder des Speicher-Bytes. Dieser Befehl bietet die gleichen Speicher-Adressier-Optionen wie der ADC-Befehl. Das erste Byte des Objektkodes wählt die Adressier-Art wie folgt:



Bit-Wert für aaa	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
000	C1	Indirekt, vor-indiziert mit X	2
001	C5	Null-Seite (direkt)	2
010	C9	Unmittelbar	2
011	CD	Absolut (direkt)	3
100	D1	Indirekt, nach-indiziert mit Y	2
101	D5	Null-Seite indiziert mit X	2
110	D9	Absolut indiziert mit Y	3
111	DD	Absolut indiziert mit X	3

Wir wollen den CMP-Befehl mit vor-indizierter indirekter Adressierung (unter Verwendung des Indexregisters X) illustrieren. Schlagen Sie die Besprechung der Adressier-Arten und der anderen Befehle für Beispiele weiterer Adressier-Arten nach.

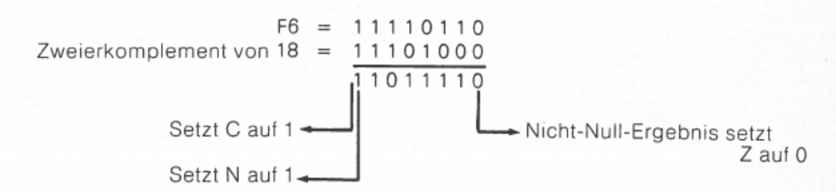


Subtrahiere den Inhalt des gewählten Speicher-Bytes vom Inhalt des Akkumulators und setze den Negativ-, Null- und Übertrags-Status, um das Ergebnis der Subtraktion wiederzugeben. Es sei angenommen $xx = FF_{16}$, $yy = 18_{16}$, $rr = 20_{16}$, $cc = 23_{16}$, $(0043_{16}) = 6D_{16}$ und $(0044_{16}) = 15_{16}$. Beachten Sie, daß 0043 = rr + cc und wir angenommen haben, daß $(156D_{16}) = yy = 18_{16}$.

Nachdem der Befehl

CMP ($23,X)

ausgeführt wurde, wird der Akkumulator noch $F6_{16}$ und der Speicherplatz $156D_{16}$ noch 18_{16} enthalten, die Status werden jedoch wie folgt geändert sein:



Beachten Sie, daß C gleich dem sich ergebenden Übertrag ist, und nicht seinem Komplement, wie es für manche andere Mikroprozessoren gilt. Daher ist C = 0, wenn ein "Borgen" erforderlich ist, und C = 1, wenn kein Borgen erforderlich war.

Vergleichs-Befehle werden häufig zum Setzen von Status vor der Ausführung von bedingten Verzweigungs-Befehlen verwendet.

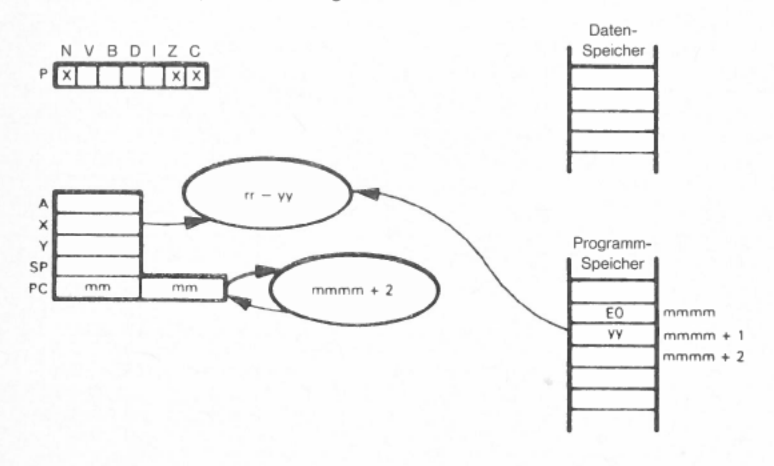
CPX – COMPARE INDEX REGISTER X WITH MEMORY (VERGLEICHE INDEXREGISTER X MIT SPEICHER)

Dieser Befehl ist der gleiche wie CMP mit der Ausnahme, daß das Speicherbyte vom Indexregister X anstatt vom Akkumulator subtrahiert wird. Die einzigen zulässigen Adressier-Arten sind die unmittelbare, Null-Seiten- (direkte) und absolute (direkte) Adressierung. Das erste Byte des Objektcodes wählt die Adressier-Art wie folgt:



Bit-Wert für aaa	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
00	E0	Unmittelbar	2
01	E4	Null-Seite (direkt)	2
10		Verwendet für INX-Befehl	
11	EC	Absolut (direkt)	3

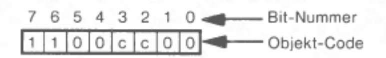
Wir wollen den CPX-Befehl mit unmittelbarer Adressierung illustrieren. Schlagen Sie die Besprechung der Adressier-Arten und weitere arithmetische und logische Befehle für Beispiele der übrigen Adressier-Arten nach.



Subtrahiere den Inhalt des gewählten Speicherbytes vom Inhalt des Indexregisters X. Der Negativ-, Null- und Übertrags-Status gibt das Ergebnis der Subtraktion in gleicher Weise wieder, wie für den CMP-Befehl gezeigt wurde.

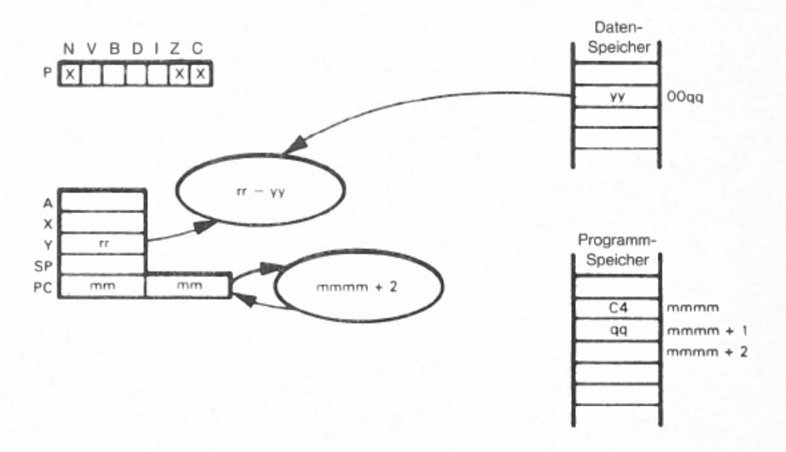
CPY – COMPARE INDEX REGISTER Y WITH MEMORY (VERGLEICHE INDEXREGISTER Y MIT SPEICHER)

Dieser Befehl ist der gleiche wie CMP mit der Ausnahme, daß das Speicherbyte vom Indexregister Y anstatt vom Akkumulator subtrahiert wird. Die einzigen zulässigen Adressier-Arten sind die unmittelbare, Null-Seiten- (direkte) und absolute (direkte) Adressierung. Das erste Byte des Objektcodes wählt die Adressier-Art wie folgt:



Bit-Wert für aaa	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
00	C0	Unmittelbar	2
01	C4	Null-Seite (direkt)	2
10		Verwendet für INY-Befehl	
11	CC	Absolut (direkt)	3

Wir wollen den CPY-Befehl mit Null-Seiten- (direkter) Adressierung illustrieren. Schlagen Sie die Besprechung der Adressier-Arten und weitere arithmetische und logische Befehle für Beispiele der anderen Adressier-Arten nach.



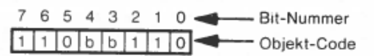
Subtrahiere den Inhalt des gewählten Speicherbytes vom Inhalt des Indexregisters Y. Das Negativ-, Null- und Übertrags-Flag geben das Ergebnis der Subtraktion auf die gleiche Weise wieder, wie für den CMP-Befehl gezeigt wurde.

DEC – DECREMENT MEMORY (BY 1) (DEKREMENTIERE SPEICHER (UM 1))

Dieser Befehl dekrementiert den Inhalt des gewählten Speicherplatzes um 1. Der DEC-Befehl verwendet vier Möglichkeiten der Datenspeicher-Adressierung:

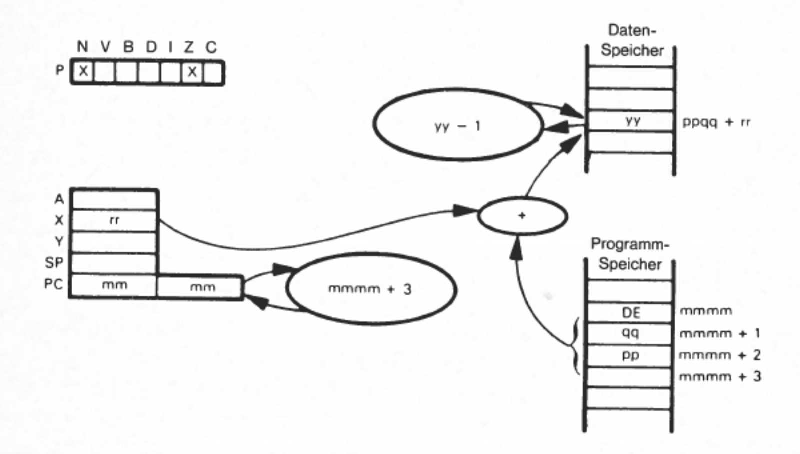
1) Null-Seite (direkt) – DEC addr
2) Absolut (direkt) – DEC addr16
3) Null-Seite indiziert mit Indexregister X – DEC addr,X
4) Absolut indiziert mit Indexregister X – DEC addr16,X

Das erste Byte des Objektcodes bestimmt die gewählte Adressier-Art wie folgt:



Bit-Wert für aaa	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
00	C6	Null-Seite (direkt)	2
01	CE	Absolut (direkt)	3
10	D6	Null-Seite indiziert mit X	2
11	DE	Absolut indiziert mit X	3

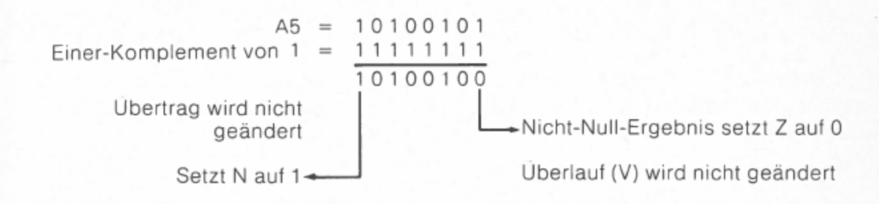
Wir wollen den DEC-Befehl mit absoluter indizierter Adressierung zeigen. Die anderen Adressier-Arten sind an anderer Stelle zu finden.



Dekrementiere den Inhalt des spezifizierten Speicherbytes.
Wenn $yy = A5_{16}$, $ppqq = 0100_{16}$ und $rr = 0A_{16}$, dann wird nach der Ausführung des Befehls

DEC $0100,X

der Inhalt des Speicherplatzes $010A_{16}$ auf $A4_{16}$ geändert.

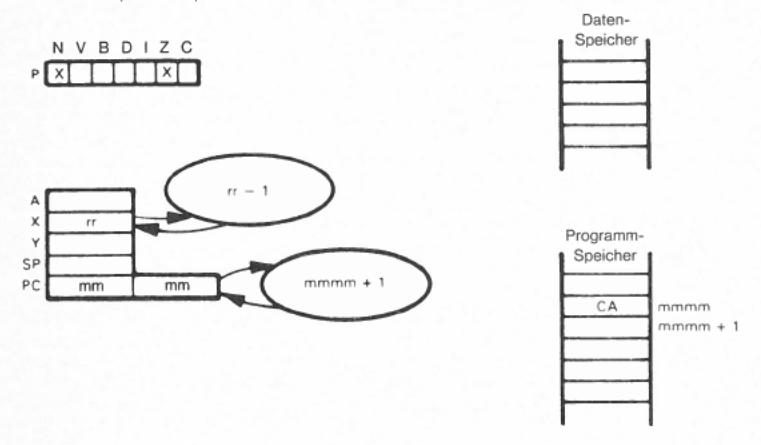


DEX – DECREMENT INDEX REGISTER X (BY 1) (DEKREMENTIERE INDEXREGISTER X (UM 1))

Dieser Befehl dekrementiert den Inhalt des Indexregisters X um 1. Der Null- und Negativ-Status werden beeinflußt.



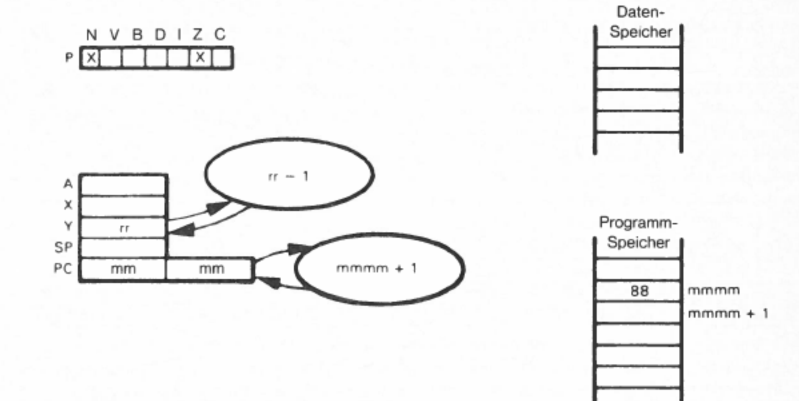
Die Auswirkungen dieses Befehls sind die gleichen wie jene von DEC mit der Ausnahme, daß der Inhalt des Indexregisters X dekrementiert wird, anstatt der Inhalt eines Speicherplatzes.



DEY – DECREMENT INDEX REGISTER Y (BY 1) (DEKREMENTIERE INDEXREGISTER Y (UM 1))

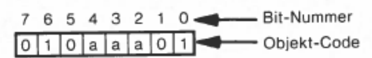
Dieser Befehl dekrementiert den Inhalt des Indexregisters Y um 1. Der Null- und Negativ-Status werden beeinflußt, so wie dies bei DEC und DEX der Fall war.





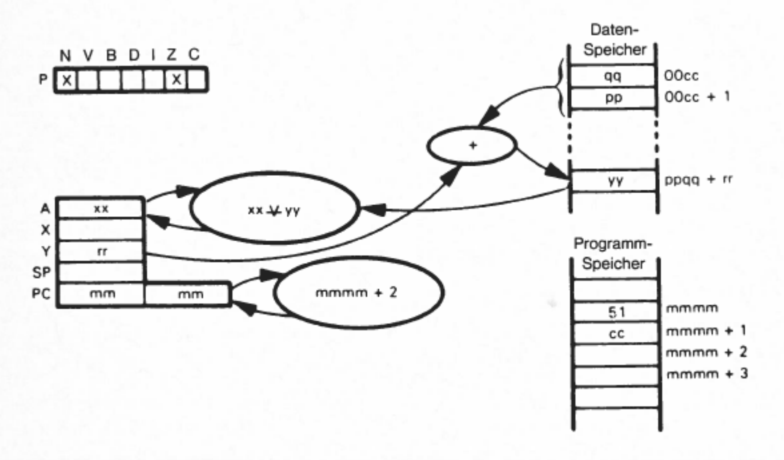
EOR – EXCLUSIVE-OR ACCUMULATOR WITH MEMORY (EXKLUSIV-ODERiere AKKUMULATOR MIT SPEICHER)

Bildet das logische Exklusiv-ODER des Inhalts des Akkumulators mit dem Inhalt eines gewählten Speicherbytes. Dieser Befehl bietet die gleichen Adressier-Optionen wie der ADC-Befehl. Das erste Byte des Objektcodes wählt die Adressier-Art wie folgt:



Bit-Wert für aaa	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
000	41	Indirekt, vor-indiziert mit X	2
001	45	Null-Seite (direkt)	2
010	49	Unmittelbar	2
011	4D	Absolut (direkt)	3
100	51	Indirekt, nach-indiziert mit Y	2
101	55	Null-Seite indiziert mit X	2
110	59	Absolut indiziert mit Y	3
111	5D	Absolut indiziert mit X	3

Wir wollen den EOR-Befehl mit nach-indizierter, indirekter Adressierung (unter Verwendung des Indexregisters Y) illustrieren. Schlagen Sie die Besprechung der Adressier-Arten und anderer arithmetischer und logischer Befehle für Beispiele der anderen Adressier-Arten nach.



Exklusiv-ODERiere logisch den Inhalt des Akkumulators mit dem Inhalt des gewählten Speicherplatzes, wobei beide Operanden als einfache Binärdaten behandelt werden. Es sei angenommen, daß $xx = 43_{16}$ und $yy = A0_{16}$. Nachdem der Befehl

EOR ($40,Y)

ausgeführt wurde, wird der Akkumulator 43_{16} enthalten. Wir nehmen auch an, daß $rr = 10_{16}$, $qq = (40_{16}) = 1E_{16}$, $pp = (41_{16}) = 25_{16}$ und $(251E_{16}) = yy = A0_{16}$.



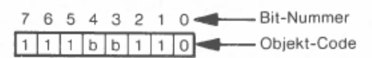
EOR wird zum Testen auf Änderungen des Bit-Status verwendet. Beachten Sie auch, daß der Befehl EOR #$FF den Inhalt des Akkumulators komplementiert, wobei er jedes 1-Bit in eine 0 und jedes 0-Bit in eine 1 verwandelt.

INC – INCREMENT MEMORY (BY 1) (INKREMENTIERE SPEICHER (UM 1))

Dieser Befehl inkrementiert den Inhalt eines gewählten Speicherplatzes um 1. Der INC-Befehl verwendet vier Arten der Datenspeicher-Adressierung:

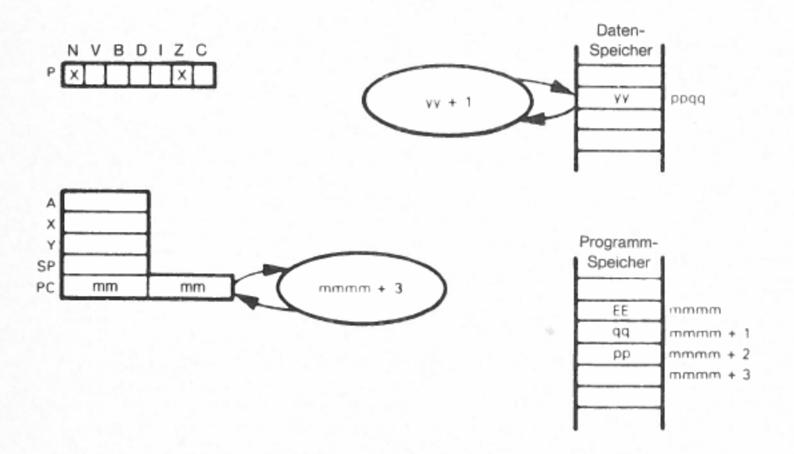
1) Null-Seite (direkt) – INC addr
2) Absolut (direkt) – INC addr16
3) Null-Seite indiziert mit Indexregister X – INC addr,X
4) Absolut indiziert mit Indexregister X – INC addr16,X

Das erste Byte des Objektcodes bestimmt wie folgt, welche Adressier-Art gewählt wurde:



Bit-Wert für aaa	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
00	E6	Null-Seite (direkt)	2
01	EE	Absolut (direkt)	3
10	F6	Null-Seite indiziert mit X	2
11	FE	Absolut indiziert mit X	3

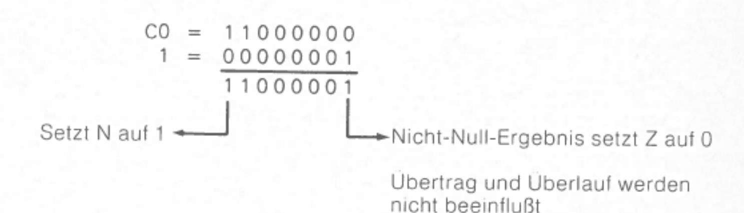
Wir wollen den INC-Befehl mit absoluter (direkter) Adressierung illustrieren. Die übrigen Adressier-Arten sind an anderer Stelle gezeigt.



Inkrementiere das gewählte Speicherbyte.
Wenn pp = 01_{16}, qq = $A2_{16}$ und yy = $C0_{16}$, dann wird nach der Ausführung eines Befehls

INC $01A2

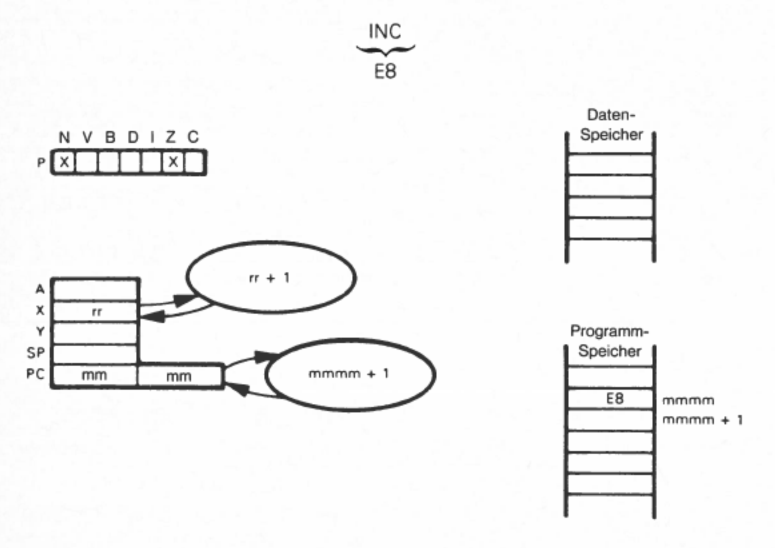
der Inhalt des Speicherplatzes $01A2_{16}$ auf $C1_{16}$ inkrementiert.



Der INC-Befehl kann zur Bildung eines Zählers in einer Vielzahl von Anwendungen eingesetzt werden, wie etwa das Zählen der Häufigkeit des Auftretens eines Vorganges oder des Spezifizierens, wie oft eine entsprechende Aufgabe auszuführen ist.

INX – INCREMENT INDEX REGISTER X (BY 1) (INKREMENTIERE INDEXREGISTER X (UM 1))

Dieser Befehl inkrementiert den Inhalt des Indexregisters X um 1. Der Null- und Negativ-Status werden so wie beim INC-Befehl beeinflußt.



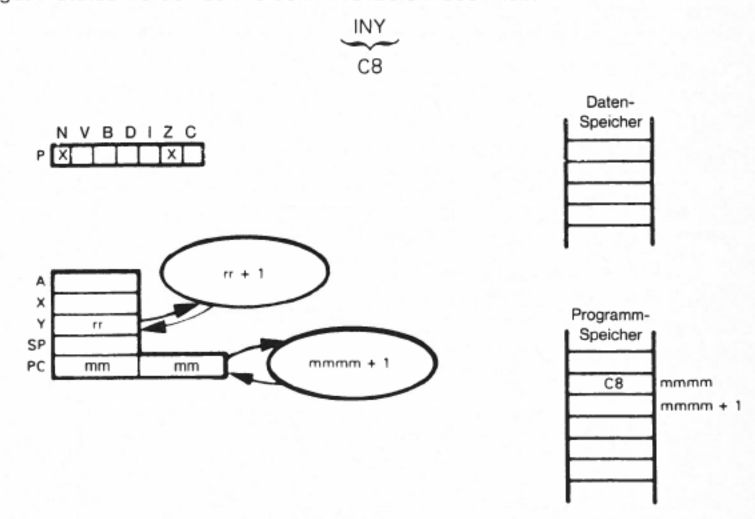
Addiere 1 zum Inhalt des Indexregisters X und setze die Null- und Negativ-Flags entsprechend dem Ergebnis. Es sei angenommen, daß das Indexregister X den Wert $7A_{16}$ enthält. Nachdem der Befehl

INX

ausgeführt wurde, wird das Indexregister $7B_{16}$ enthalten, der Null-Status wird gelöscht werden, da das Ergebnis nicht Null ist, und der Negativ-Status wird gelöscht sein, da das Ergebnis 0 in seinem höchstwertigen Bit enthält.

INY – INCREMENT INDEX REGISTER Y (BY 1) (INKREMENTIERE INDEXREGISTER Y (UM 1))

Dieser Befehl inkrementiert den Inhalt des Indexregisters Y um 1. Der Null- und Negativ-Status werden so wie beim INC-Befehl beeinflußt.



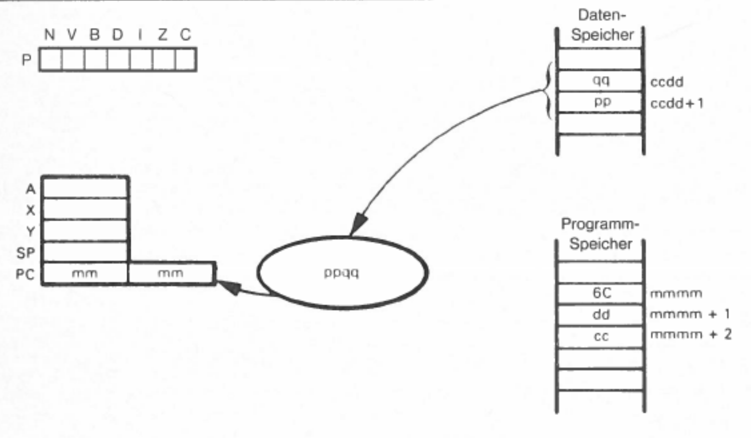
Addiere 1 zum Inhalt des Indexregisters Y und setze das Null- und Negativ-Flag entsprechend dem Ergebnis. Es sei angenommen, daß das Indexregister Y den Wert $0C_{16}$ enthält. Nachdem der Befehl INY ausgeführt wurde, wird das Indexregister Y den Wert $0D_{16}$ enthalten, der Null-Status wird gelöscht sein, da das Ergebnis nicht Null ist, und der Negativ-Status wird gelöscht sein, da das Ergebnis 0 in seinem höchstwertigen Bit besitzt.

JMP – JUMP VIA ABSOLUTE OR INDIRECT ADDRESSING (SPRINGE MITTELS ABSOLUTER ODER INDIREKTER ADRESSIERUNG)

Dieser Befehl wird unter Verwendung der indirekten Adressierung illustriert werden. Beachten Sie, daß es der einzige Befehl ist, der die echte indirekte Adressierung besitzt. Das erste Byte des Objektcodes bestimmt die Adressier-Art wie folgt:



Bit-Wert für y	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
0	4C	Absolut (direkt)	3
1	6C	Indirekt	3



Springe zu dem Befehl, der durch den Operanden spezifiziert wird, durch Laden der Adresse von den gewählten Speicherbytes in den Befehlszähler.
In der folgenden Befehls-Sequenz:

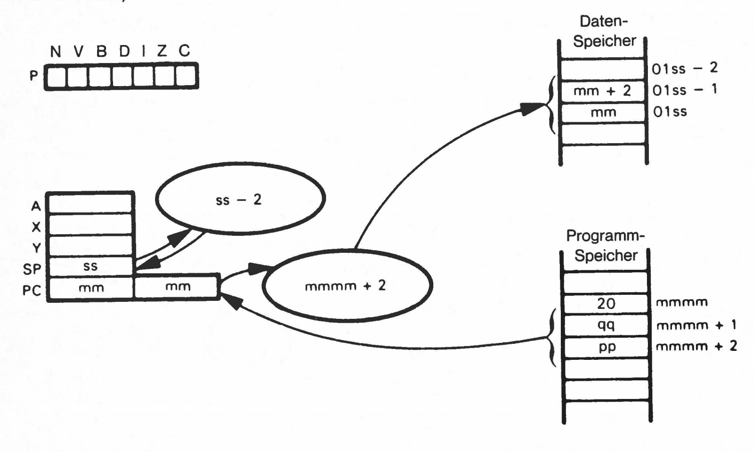
```
CLC
LDA     #BASEL  ;BERECHNE LSB'S DER BESTIMMUNGS-ADRESSE
ADC     INDXL
STA     JADDR
LDA     #BASEU  ;BERECHNE MSB'S DER BESTIMMUNGS-ADRESSE
ADC     INDXU
STA     JADDR+1
JMP     (JADDR)  ;TRANSFERIERE STEUERUNG ZUR ZIEL-ADRESSE
```


Der JMP-Befehl wird einen indizierten Sprung relativ zur 16-Bit-Adresse ausführen, die aus BASEU (8 MSBs) und BASEL (8 LSBs) besteht. Es wird hier angenommen, daß der Index 16 Bits lang ist und anfangs bei den Adressen INDXL (8 LSBs) und INDXU (8 MSBs) gespeichert ist. Die Adressen, die dem Beginn der Tabelle folgen, könnten dann absolute JMP-Befehle enthalten, die die Steuerung zu den entsprechenden Routinen transferieren.

Der JMP-Befehl kann auch die absolute (direkte) Adressier-Art verwenden. In diesem Fall wird das zweite Byte des Befehls in das niederwertige Byte des Befehlszählers und das dritte Byte des Befehls in das höherwertige Byte des Befehlszählers geladen. Die Befehlsausführung wird von dieser Adresse ab fortgesetzt.

JSR – JUMP TO SUBROUTINE (SPRINGE ZU UNTERPROGRAMM)

Dieser Befehl bringt den Inhalt des Befehlszählers auf den Stapel und überträgt dann die Steuerung dem spezifizierten Befehl. Es ist nur absolute (direkte) Adressierung gestattet. Beachten Sie, daß der Stapelzeiger nach der Speicherung jedes Datenbytes inkrementiert wird und daß der Wert des Befehlszählers, der aufbewahrt wird, die Adresse des letzten (dritten) Bytes des JSR-Befehls darstellt, das heißt, der anfängliche Wert des Befehlszählers plus 2. Erinnern Sie sich auch daran, daß der Stapel im Speicher abwärts wächst und daß die höchstwertige Hälfte des Befehlszählers zuerst gespeichert wird und daher bei der höheren Adresse endet (in der gewöhnlichen Adressier-Form des 6502).



Der Befehlszähler wird um 2 inkrementiert und dann auf den Stapel gebracht. Der Stapelzeiger wird so eingerichtet, daß er zum nächsten leeren Speicherplatz im Stapel zeigt. Der Adressen-Teil des Befehls wird dann in den Befehlszähler gespeichert und die Ausführung wird ab diesem Punkt fortgesetzt.

Es sei angenommen mmmm = $E34F_{16}$ und ss = $E3_{16}$. Dann wird nach der Ausführung des Befehls

JSR $E100

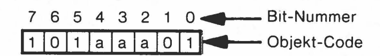
der Befelszähler den Wert $E100_{16}$ enthalten, der Stapelzeiger wird $E1_{16}$ enthalten und die Stapelplätze werden wie folgt lauten:

(01ss) = (01E3) = PC(HI) = E3
(01ss – 1) = (01E2) = PC(LO) = 51_{16}

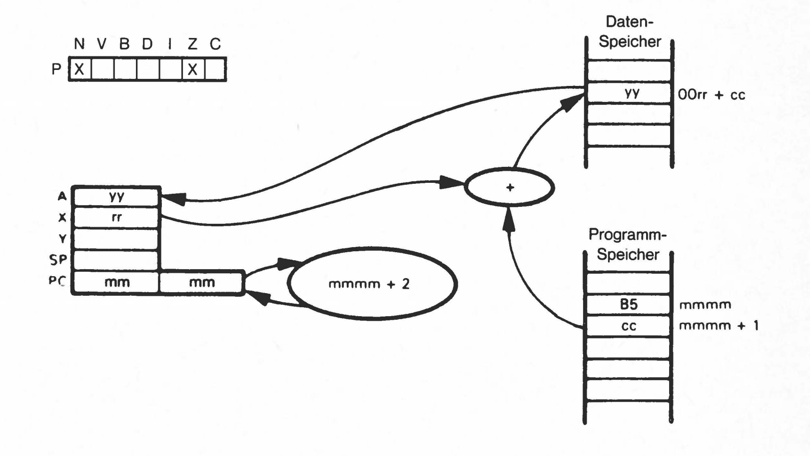
Der nächste auszuführende Befehl wird derjenige bei der Speicher-Adresse $E100_{16}$ sein.

LDA – LOAD ACCUMULATOR FROM MEMORY (LADE DEN AKKUMULATOR VOM SPEICHER)

Lädt den Inhalt des gewählten Speicherbytes in den Akkumulator. Dieser Befehl bietet die gleichen Speicher-Adressiermöglichkeiten, wie der ADC-Befehl und wird unter Verwendung der indizierten Nullseiten-Adressierung mit dem Indexregister X illustriert. Schlagen Sie die Besprechung der Adressier-Arten und der anderen arithmetischen und logischen Befehle für Beispiele der übrigen Adressier-Arten nach. Das erste Byte des Objektcodes wählt die Adressier-Art wie folgt:



Bit-Wert für aaa	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
000	A1	Indirekt, vor-indiziert mit X	2
001	A5	Null-Seite (direkt)	2
010	A9	Unmittelbar	2
011	AD	Absolut (direkt)	3
100	B1	Indirekt, nach-indiziert mit Y	2
101	B5	Null-Seite indiziert mit X	2
110	B9	Absolut indiziert mit Y	3
111	BD	Absolut indiziert mit X	3



Lade den Inhalt des gewählten Speicherbytes in den Akkumulator.

Es sei angenommen, daß das Indexregister X den Wert 10_{16} enthält und cc = 43_{16}. Wenn der Speicherplatz 0053_{16} den Wert AA_{16} enthält, dann wird nach Ausführung von

LDA $43,X

der Akkumulator den Wert AA_{16} enthalten.

AA = 1 0 1 0 1 0 1 0

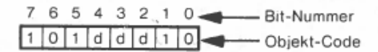
1 setzt N auf 1 ← (Bit 7) — (Bit 0) → Nicht-Null-Ergebnis setzt Z auf 0

LDX – LOAD INDEX REGISTER X FROM MEMORY (LADE INDEXREGISTER X VOM SPEICHER)

Lädt den Inhalt des gewählten Speicherbytes in das Indexregister X. Die zulässigen Adressier-Arten sind:

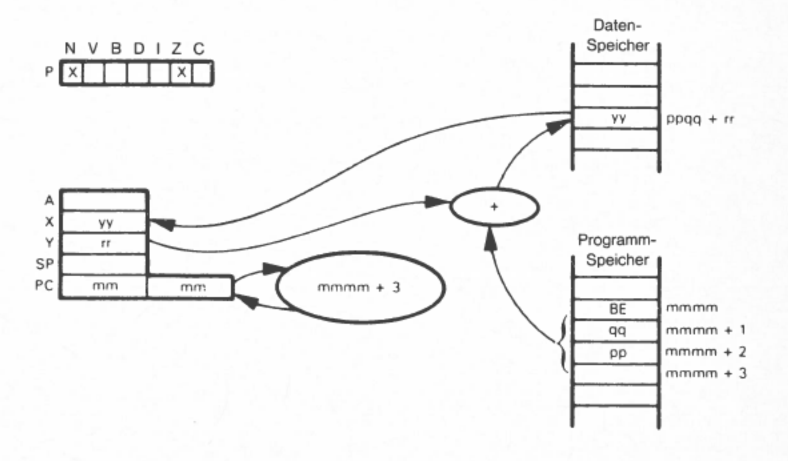
1) Unmittelbar – LDX data
2) Absolut (direkt) – LDX addr16
3) Null-Seite (direkt) – LDX addr
4) Absolut indiziert mit Y – LDX addr16,Y
5) Null-Seite indiziert mit Y – LDX addr,Y

Beachten Sie, daß es keine indizierten Adressier-Arten mit dem Indexregister X gibt, sowie keine Nach-Indizierung.
Das erste Byte des Objektcodes wählt die Adressier-Art wie folgt:



Bit-Wert für aaa	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
000	A2	Unmittelbar	2
001	A6	Null-Seite (direkt)	2
010	AA	Verwendet für TAX-Befehl	
011	AE	Absolut (direkt)	3
100	B2	Nicht verwendet	
101	B6	Null-Seite indiziert mit Y	2
110	BA	Verwendet für TSX-Befehl	
111	BE	Absolut indiziert mit Y	3

Wir wollen den LDX-Befehl mit absoluter indizierter Adressierung unter Verwendung des Indexregisters Y illustrieren. Schlagen Sie die Besprechung der Adressier-Arten und anderer arithmetischer und logischer Befehle für Beispiele der übrigen Adressier-Arten nach.



Lade den Inhalt des gewählten Speicherbytes in das Indexregister X. Es sei angenommen, daß das Indexregister Y den Wert 28_{16} enthält, ppqq = $2E1A_{16}$ und yy = ($2E42_{16}$) = $4F_{16}$. Dann wird nach der Ausführung des Befehls

```
LDX   $3E1A,Y
```

das Indexregister X den Wert $4F_{16}$ enthalten.

4F = 0 1 0 0 1 1 1 1

0 setzt N auf 0 ← (Bit 7) — (Ergebnis) → Nicht-Null-Ergebnis setzt Z auf 0

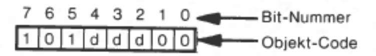
LDY – LOAD INDEX REGISTER Y FROM MEMORY (LADE INDEXREGISTER Y VOM SPEICHER)

Lädt den Inhalt des gewählten Speicherbytes in das Indexregister Y. Die zulässigen Adressier-Arten sind:

1) Unmittelbar – LDY data
2) Absolut (direkt) – LDY addr16
3) Null-Seite (direkt) – LDY addr
4) Absolut indiziert mit X – LDY addr16,X
5) Null-Seite indiziert mit X – LDY addr,X

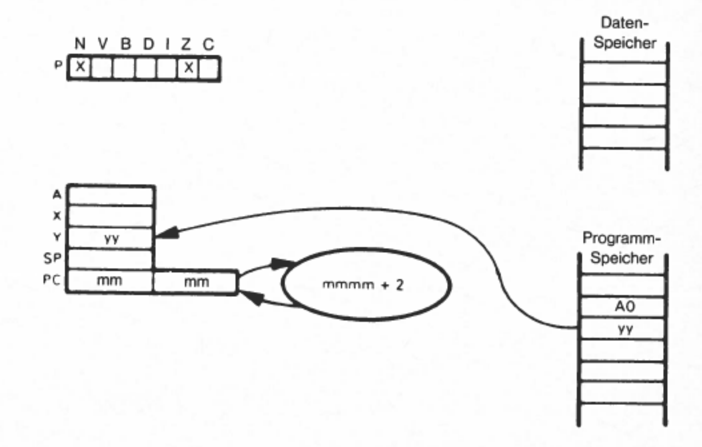
Beachten Sie, daß es weder indizierte Adressier-Arten mit dem Indexregister Y gibt, noch irgendwelche Vor-Indizierung.

Das erste Byte des Objektcodes wählt die Adressier-Art wie folgt:



Bit-Wert für ddd	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
000	A0	Unmittelbar	2
001	A4	Null-Seite (direkt)	2
010	A8	Verwendet für TAY-Befehl	
011	AC	Absolut (direkt)	3
100	B0	Verwendet für BCS-Befehl	
101	B4	Null-Seite indiziert durch X	2
110	B8	Verwendet für CLV-Befehl	
111	BC	Absolut indiziert durch X	3

Wir wollen den LDY-Befehl mit unmittelbarer Adressierung illustrieren. Schlagen Sie die Besprechung der Adressier-Arten und anderer arithmetischer und logischer Befehle für Beispiele der übrigen Adressier-Arten nach.



Lade den Inhalt des gewählten Speicherbytes in das Indexregister Y. Es sei angenommen, daß yy = 00_{16}, so daß nach der Ausführung des Befehls

LDY #0

das Indexregister Y den Wert 0 enthalten wird.

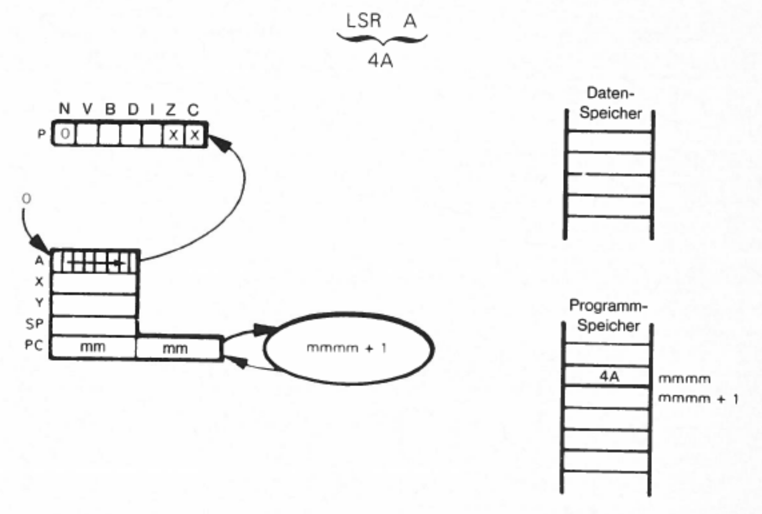
```
              00 = 00000000
                   |      |
0 setzt N auf 0<---+      +--->Null-Ergebnis setzt Z auf 1
```


LSR – LOGICAL SHIFT RIGHT OF ACCUMULATOR OR MEMORY (LOGISCHE RECHTS-VERSCHIEBUNG DES AKKUMULATORS ODER SPEICHERS)

Dieser Befehl führt eine logische Rechts-Verschiebung um ein Bit des Akkumulators oder des gewählten Speicherbytes aus.

Zuerst sei die Verschiebung des Akkumulators betrachtet.

LSR A
4A



Schiebe den Inhalt des Akkumulators um ein Bit nach rechts. Schiebe das niederwertige Bit in den Übertrags-Status. Schiebe eine Null in das hochwertige Bit.

Der Akkumulator enthalte angenommen $7A_{16}$. Nachdem der Befehl

LSR A

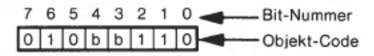
ausgeführt wurde, wird der Akkumulator den Wert $3D_{16}$ enthalten und der Übertragungs-Status wird auf Null gesetzt sein.

```
                    Akkumulator     Übertrag
               0--->01111010------>X
                    00111101       0
                    |      |
LSR setzt N immer <-+      +-----> Nicht-Null-Ergebnis setzt Z auf 0
          auf 0
```

Vier Arten der Adressierung des Datenspeichers sind mit dem LSR-Befehl verfügbar. Diese sind:

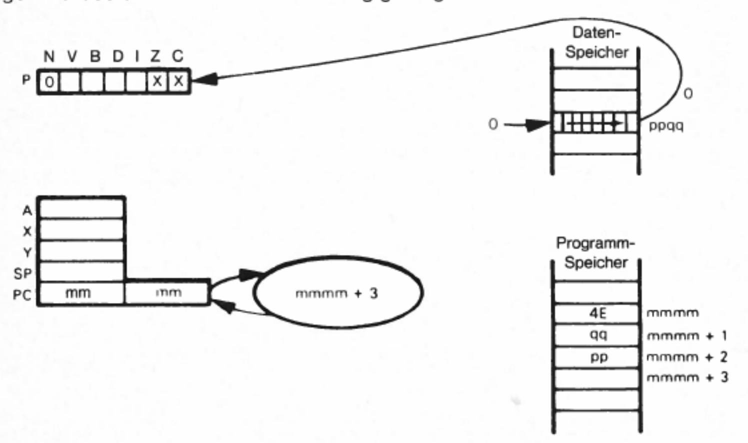
1) Null-Seite (direkt) – LSR addr
2) Absolut (direkt) – LSR addr16
3) Null-Seite indiziert mit Indexregister X – LSR addr,X
4) Absolut indiziert mit Indexregister X – LSR addr16,X

Das erste Byte des Objektcodes bestimmt die gewählte Adressier-Art wie folgt:



Bit-Wert für aaa	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
00	46	Null-Seite (direkt)	2
01	4E	Absolut (direkt)	3
10	56	Null-Seite indiziert mit X	2
11	5E	Absolut indiziert mit X	3

Wir wollen den LSR-Befehl mit absoluter (direkter) Adressierung illustrieren. Die übrigen Adressier-Arten sind anderweitig gezeigt.

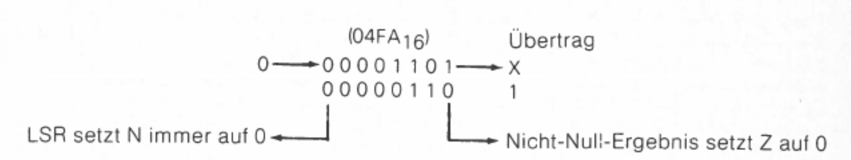


Verschiebe den Inhalt des gewählten Speicherbytes logisch um ein Bit nach rechts.

Es sei angenommen ppqq = $04FA_{16}$ und der Inhalt des Speicherplatzes $04FA_{16}$ sei $0D_{16}$. Nachdem der Befehl

LSR $04FA

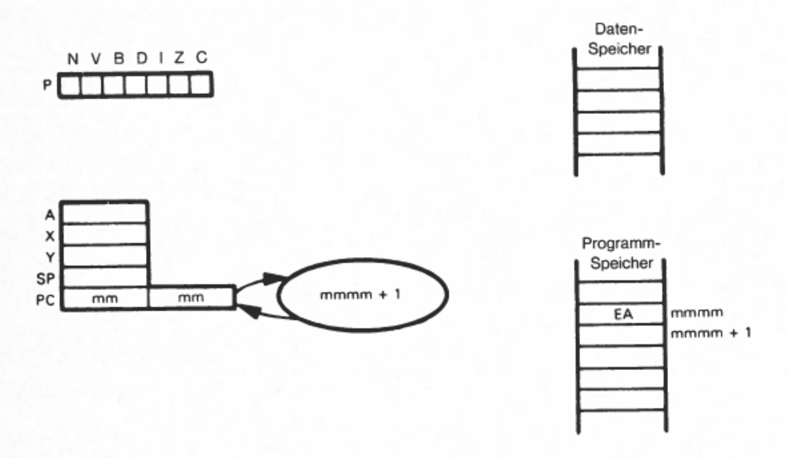
ausgeführt wurde, wird der Übertags-Status 1 sein und der Inhalt des Speicherplatzes $04FA_{16}$ wird 06_{16} sein.



NOP – NO OPERATION (KEINE OPERATION)

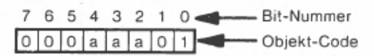


Dies ist ein Ein-Byte-Befehl, der nichts weiter ausführt, als den Befehlszähler zu inkrementieren. Dieser Befehl gestattet Ihnen, eine Markierung (label) einem Objektprogramm-Byte zu geben, oder eine Verzögerung fein einzustellen (jeder NOP-Befehl addiert zwei Taktzyklen) und Befehls-Bytes zu ersetzen, die nicht länger infolge von Korrekturen oder Änderungen erforderlich sind. NOPs können auch zum Ersetzen von Befehlen (wie etwa JSRs) verwendet werden, die Sie vielleicht in Fehlersuch-Läufen nicht wollen. NOP wird nicht sehr häufig in endgültigen Programmen verwendet, ist jedoch häufig bei der Fehlersuche und beim Testen von Nutzen.



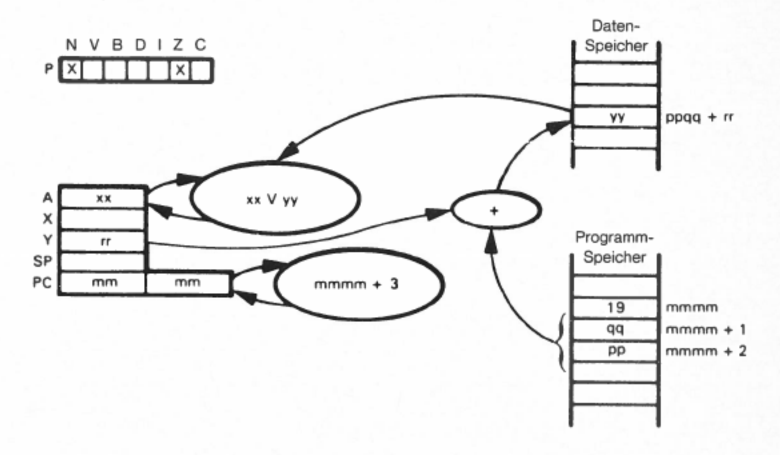
ORA – LOGICALLY OR MEMORY WITH ACCUMULATOR (ODERiere SPEICHER MIT AKKUMULATOR LOGISCH)

Dieser Befehl ODERiert logisch den Inhalt eines Speicherplatzes mit dem Inhalt des Akkumulators. Dieser Befehl bietet die gleichen Speicher-Adressiermöglichkeiten wie der ADC-Befehl. Das erste Byte des Objektcodes wählt die Adressier-Art wie folgt:



Bit-Wert für aaa	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
000	01	Indirekt, vor-indiziert mit X	2
001	05	Null-Seite (direkt)	2
010	09	Unmittelbar	2
011	0D	Absolut (direkt)	3
100	11	Indirekt, nach-indiziert mit Y	2
101	15	Null-Seite indiziert mit X	2
110	19	Absolut indiziert mit Y	3
111	1D	Absolut indiziert mit X	3

Wir wollen den ORA-Befehl unter Verwendung der absoluten indizierten Adressierung mit dem Indexregister Y illustrieren. Schlagen Sie die Besprechung der Adressier-Arten und anderer arithmetischer und logischer Befehle für Beispiele für die übrigen Adressier-Arten nach.



ODERiere logisch den Inhalt des Akkumulators mit dem Inhalt des gewählten Speicherbytes, wobei beide Operanden als einfache Binärdaten behandelt werden.

Es sei angenommen ppqq = 1623_{16}, rr = 10_{16}, xx = $E3_{16}$ und yy = AB_{16}. Nach Ausführung des Befehls

ORA $1623,jY

wird der Akkumulator EB_{16} enthalten.

```
E3 = 11100011
AB = 10101011
     --------
     11101011
```

Setzt N auf 1 ← (höchstwertiges Bit) — Nicht-Null-Ergebnis setzt Z auf 0

Dies ist ein logischer Befehl, der häufig zum "Ein"-Schalten von Bits verwendet wird, das heißt, sie auf 1 zu setzen. Beispielsweise wird der Befehl

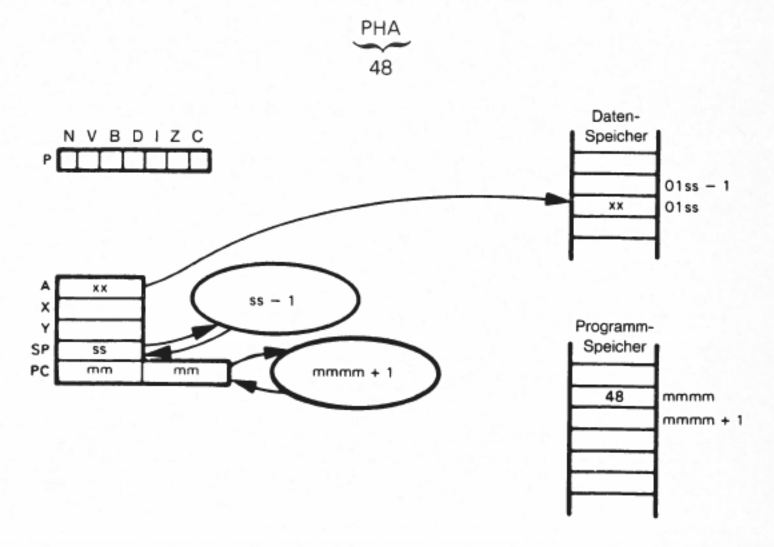
ORA #$80

das hochwertige Bit im Akkumulator ohne Bedingung auf 1 setzen.

PHA – PUSH AKKUMULATOR ONTO STACK (BRINGE AKKUMULATOR AUF DEN STAPEL)

Dieser Befehl speichert den Inhalt des Akkumulators auf die Spitze des Stapels. Der Stapelzeiger wird dann um 1 dekrementiert. Es werden keine weiteren Register oder Status beeinflußt. Beachten Sie, daß der Akkumulator in den Stapel gespeichert wird, bevor der Stapelzeiger dekrementiert wird.

PHA
48



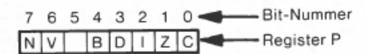
Der Akkumulator enthalte angenommen $3A_{16}$ und der Stapelzeiger $F7_{16}$. Nachdem der Befehl PHA ausgeführt wurde, wird $3A_{16}$ in den Speicherplatz $01F7_{16}$ gespeichert und der Stapelzeiger auf $F6_{16}$ geändert worden sein.

Der PHA-Befehl wird am häufigsten zur Aufbewahrung des Akkumulator-Inhaltes verwendet, bevor eine Unterbrechung bedient oder ein Unterprogramm aufgerufen wird.

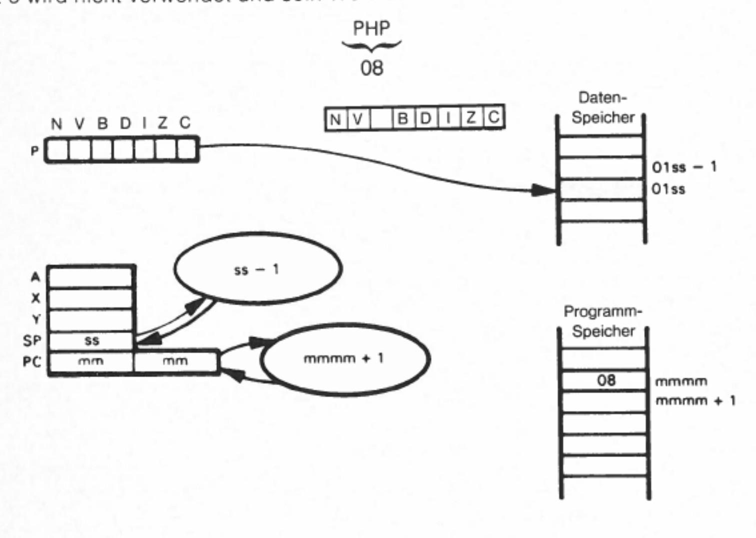
PHP – PUSH STATUS REGISTER (P) ONTO STACK (BRINGE STATUSREGISTER (P) AUF DEN STAPEL)

Dieser Befehl speichert den Inhalt des Statusregisters (P) auf die Spitze des Stapels. Der Stapelzeiger wird dann um 1 dekrementiert. Es werden keine anderen Register oder Status beeinflußt. Beachten Sie, daß das Statusregister in den Stapel gespeichert wird, bevor der Stapelzeiger dekrementiert wird.

Die Organisation des Status im Speicher ist wie folgt:



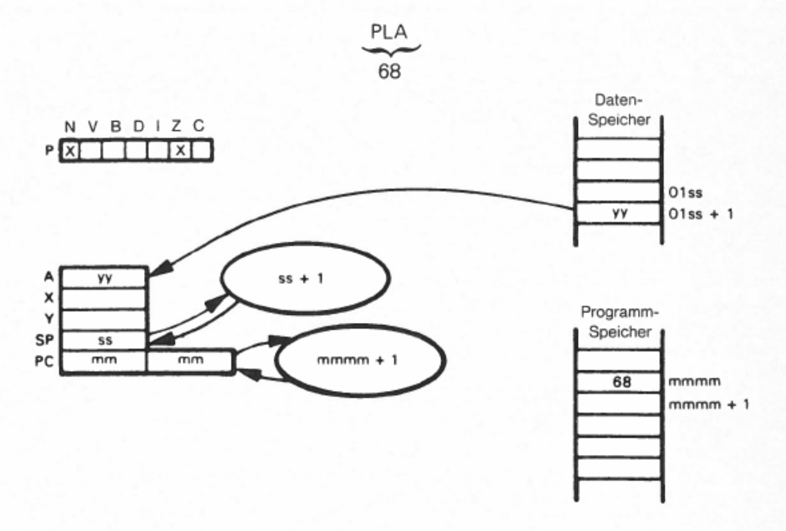
Bit 5 wird nicht verwendet und sein Wert ist willkürlich.



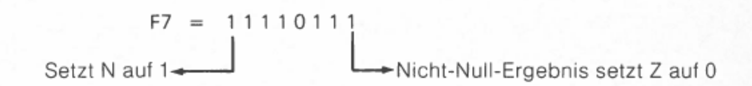
Der PHP-Befehl wird im allgemeinen zur Aufbewahrung des Inhalts des Statusregisters vor dem Aufrufen eines Unterprogramms verwendet. Beachten Sie, daß PHP nicht erforderlich ist, bevor eine Unterbrechung bedient wird, da die Unterbrechungs-Reaktion (auf IRQ oder NMI) und der BRK-Befehl automatisch den Inhalt des Statusregisters an der Spitze des Stapels aufbewahren.

PLA – PULL CONTENTS OF ACCUMULATOR FROM STACK (HOLE INHALT DES AKKUMULATORS VOM STAPEL)

Dieser Befehl inkrementiert den Stapelzeiger um 1 und lädt dann den Akkumulator von der Spitze des Stapels. Beachten Sie, daß der Stapelzeiger vor dem Laden des Akkumulators inkrementiert wird.



Der Stapelzeiger enthalte angenommen $F6_{16}$, und der Speicherplatz $01F7_{16}$ enthalte CE_{16}. Nach der Ausführung des PLA-Befehls wird der Akkumulator CE_{16} und der Stapelzeiger $F7_{16}$ enthalten.



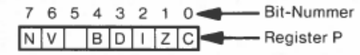
Der PLA-Befehl wird am häufigsten zur Zurückspeicherung des Akkumulator-Inhaltes verwendet, der auf den Stapel gerettet wurde, z.B. nach der Bedienung einer Unterbrechung oder nach dem Abschluß eines Unterprogramms.

PLP – PULL CONTENTS OF STATUS REGISTER (P) FROM STACK (HOLE INHALT DES STATUSREGISTERS (P) VOM STAPEL)

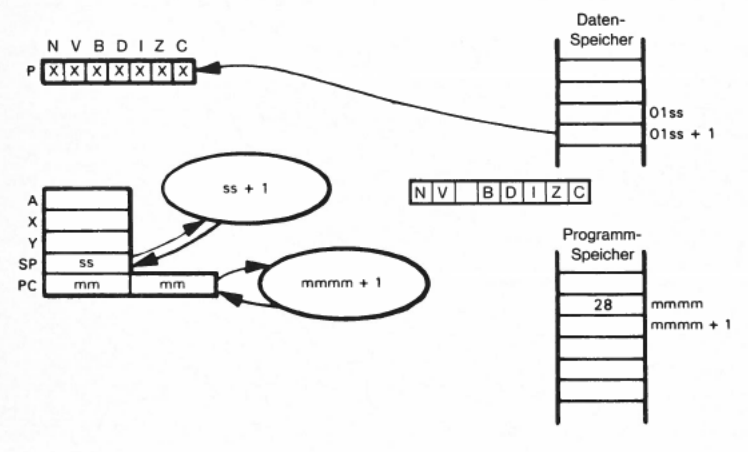
Dieser Befehl inkrementiert den Stapelzeiger um 1 und lädt dann das Statusregister (P) von der Spitze des Stapels. Es werden keine anderen Register beeinflußt, es können jedoch alle Status geändert werden. Beachten Sie, daß der Stapelzeiger vor dem Laden des Statusregisters inkrementiert wird.

PLP
28

Die Organisation des Status im Speicher sieht wie folgt aus:



Bit 5 wird nicht verwendet.



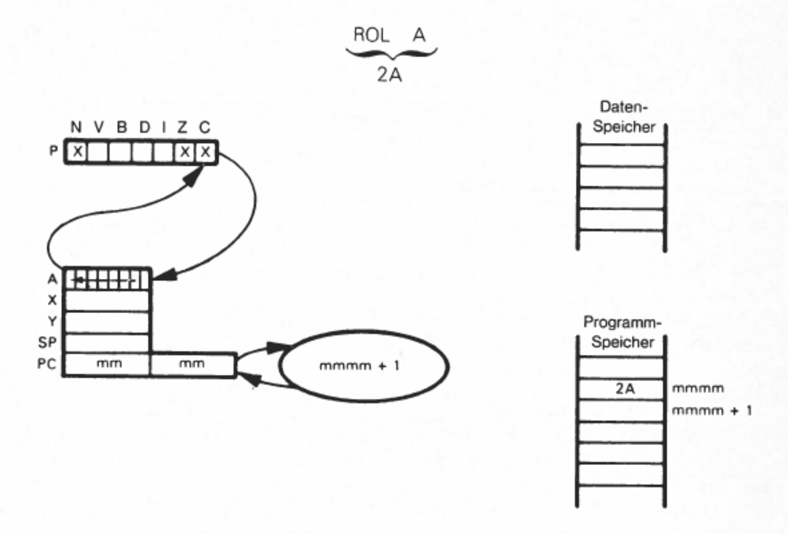
Der PLP-Befehl wird im allgemeinen zum Zurückspeichern des Inhalts des Statusregisters nach Abschluß eines Unterprogramms verwendet. Daher dient er zum Ausgleich des früher erwähnten PHP-Befehls. Beachten Sie, daß PLP nach der Bedienung einer Unterbrechung nicht erforderlich ist, da der RTI-Befehl automatisch den Inhalt des Statusregisters von der Spitze des Stapels zurückspeichert.

ROL – ROTATE ACCUMULATOR OR MEMORY LEFT THROUGH CARRY (ROTIERE AKKUMULATOR ODER SPEICHER NACH LINKS DURCH DEN ÜBERTRAG)

Dieser Befehl rotiert den Akkumulator oder das gewählte Speicherbyte um ein Bit nach links durch den Übertrag.

Zuerst sei die Rotation des Akkumulators betrachtet.

ROL A
2A

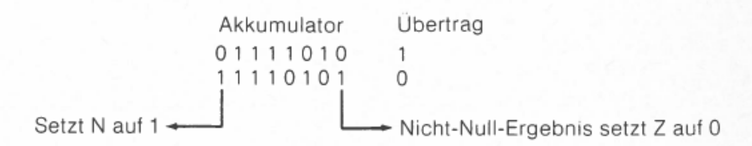


Rotiere den Inhalt des Akkumulators um ein Bit nach links durch den Übertrags-Status.

Der Akkumulator enthalte angenommen $7A_{16}$, und der Übertrags-Status wird auf 1 gesetzt. Nachdem der Befehl

ROL A

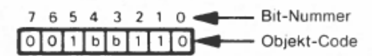
ausgeführt wurde, wird der Akkumulator $F5_{16}$ enthalten, und der Übertrags-Status wird auf Null gelöscht.



Der ROL-Befehl gestattet vier Arten der Adressierung des Datenspeichers. Diese sind:

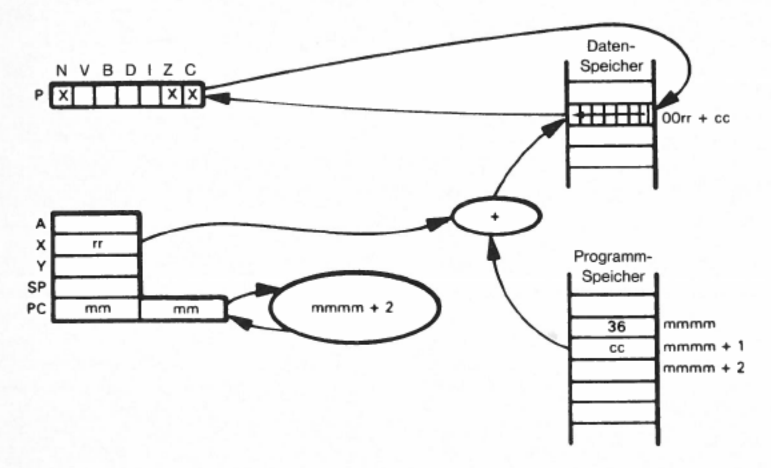
1) Null-Seite (direkt) – ROL addr
2) Absolut (direkt) – ROL addr16
3) Null-Seite indiziert mit Indexregister X – ROL addr,16
4) Absolut indiziert mit Indexregister X – ROL addr16,X

Das erste Byte des Objektcodes bestimmt, welche Adressier-Art wie folgt gewählt wurde:



Bit-Wert für aaa	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
00	26	Null-Seite (direkt)	2
01	2E	Absolut (direkt)	3
10	36	Null-Seite indiziert mit X	2
11	3E	Absolut indiziert mit X	3

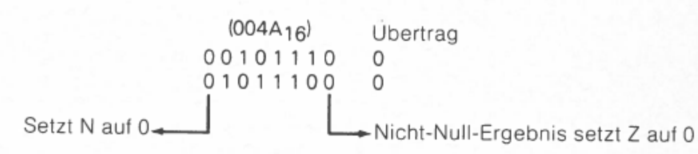
Wir wollen den ROL-Befehl mit indizierter Nullseiten-Adressierung (unter Verwendung des Indexregisters X) illustrieren. Die übrigen Adressier-Arten sind an anderer Stelle gezeigt.



Rotiere das gewählte Speicherbyte um ein Bit nach links durch den Übertrags-Status. Es sei angenommen $cc = 34_{16}$, $rr = 16_{16}$, der Inhalt des Speicherplatzes $004A_{16}$ sei $2E_{16}$ und der Übertrags-Status ist Null. Nach Ausführung eines Befehls

ROL $34,X

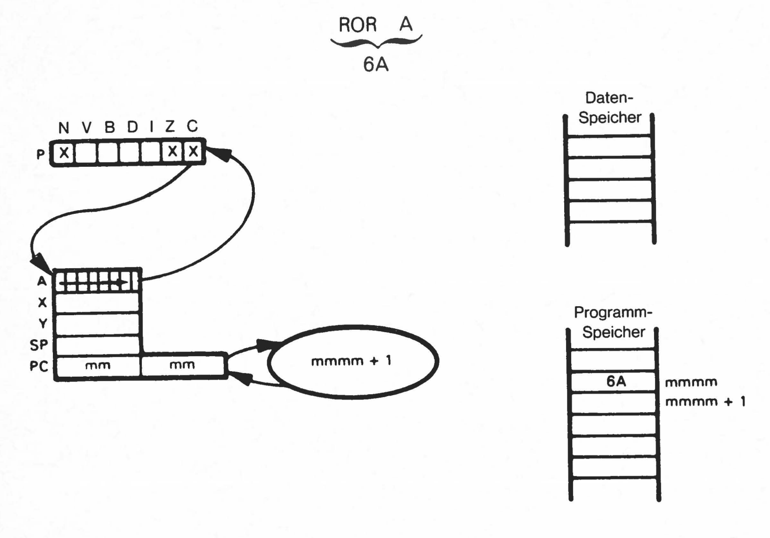
wird der Speicherplatz $004A_{16}$ den Wert $5C_{16}$ enthalten.



ROR – ROTATE ACCUMULATOR OR MEMORY RIGHT, THROUGH CARRY (ROTIERE AKKUMULATOR ODER SPEICHER NACH RECHTS DURCH DEN ÜBERTRAG)

Dieser Befehl rotiert den Akkumulator oder das gewählte Speicherbyte um ein Bit nach rechts durch den Übertrag.

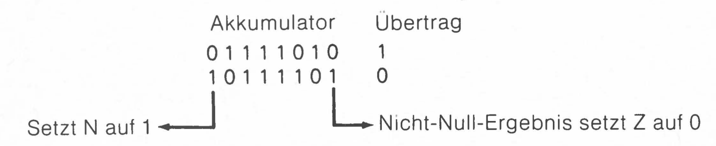
Betrachten wir zuerst die Rotation des Akkumulators.



Rotiere den Inhalt des Akkumulators um ein Bit nach rechts durch den Übertrags-Status. Der Akkumulator enthalte angenommen $7A_{16}$, und der Übertrags-Status wird auf 1 gesetzt. Die Ausführung des Befehls

ROR A

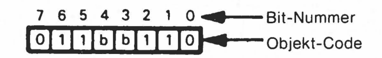
wird diese Resultate liefern: Der Akkumulator wird BD_{16} enthalten, und der Übertrags-Status wird Null sein.



Der ROR-Befehl gestattet vier Arten der Adressierung des Datenspeichers. Diese sind:

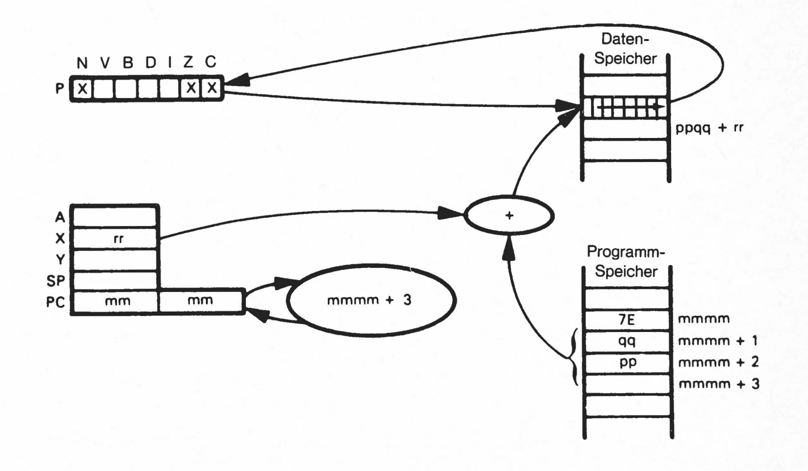
1) Null-Seite (direkt) – ROR addr
2) Absolut (direkt) – ROR addr16
3) Null-Seite indiziert mit Indexregister X – ROR addr,X
4) Absolut indiziert mit Indexregister X – ROR addr16,X

Das erste Byte des Objektcodes bestimmt, welche Adressier-Art wie folgt gewählt wurde:



Bit-Wert für aaa	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
00	66	Null-Seite (direkt)	2
01	6E	Absolut (direkt)	3
10	76	Null-Seite indiziert mit X	2
11	7E	Absolut indiziert mit X	3

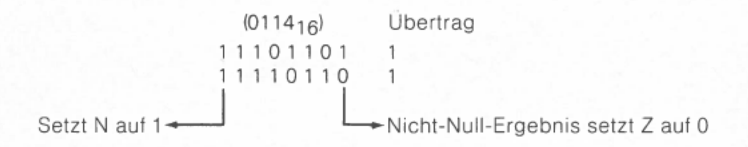
Wir wollen den ROR-Befehl mit absoluter indizierter Adressierung illustrieren (unter Verwendung des Indexregisters X). Die übrigen Adressier-Arten sind an anderer Stelle gezeigt.



Es sei angenommen rr = 14_{16}, ppqq = 0100_{16}, der Inhalt des Speicherplatzes 0114_{16} sei ED_{16} und der Übertrags-Status sei 1. Nach Ausführung eines Befehls

ROR $0100,X

wird der Übertrags-Status 1 sein, und der Speicherplatz 0114_{16} wird $F6_{16}$ enthalten.

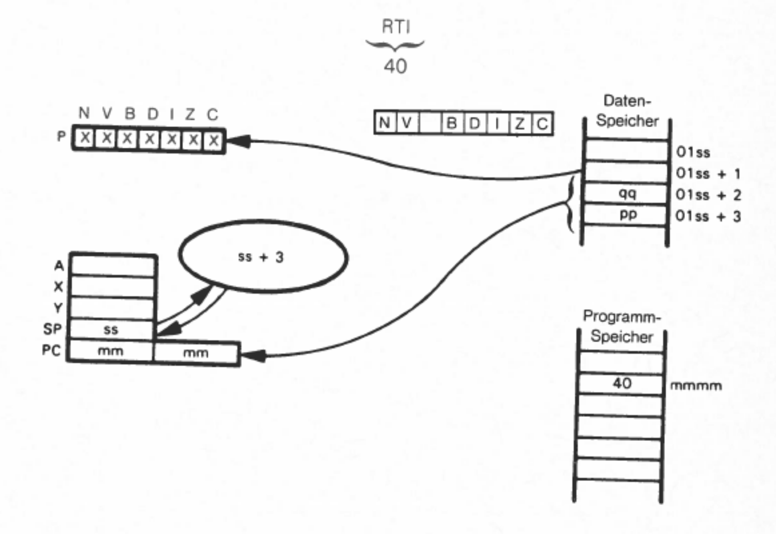


RTI – RETURN FROM INTERRUPT (KEHRE VON UNTERPROGRAMM ZURÜCK)

Holt das Statusregister (P) und den Befehlszähler von der Spitze des Stapels zurück. Die Register und die entsprechenden Speicherplätze, von denen sie geladen wurden, sind folgende, wobei angenommen wird, daß der Stapelzeiger ss bei Beginn der Befehls-Ausführung enthält:

Speicherplatz	Register
01ss+1	Statusregister (P)
01ss+2	Niederwertiges Byte des Befehlszählers
01ss+3	Hochwertiges Byte des Befehlszählers

Der abschließende Wert des Stapelzeigers ist sein Anfangswert plus 3. Die alten Werte des Statusregisters und des Befehlszählers gehen verloren.



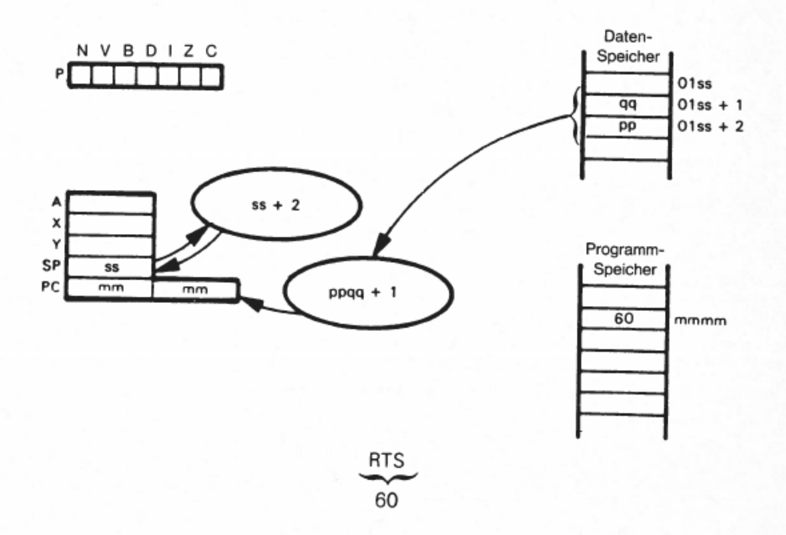
Der Stapelzeiger enthalte angenommen $E8_{16}$, der Speicherplatz $01E9_{16}$ enthalte $C1_{16}$, der Speicherplatz $01EA_{16}$ enthalte $3E_{16}$, und der Speicherplatz $01EB_{16}$ enthalte $D5_{16}$. Nachdem der Befehl RTI ausgeführt wurde, wird das Statusregister $C1_{16}$, der Stapelzeiger EB_{16} und der Befehlszähler $D53E_{16}$ enthalten (dies ist die Adresse, von wo ab die Befehls-Ausführung fortgesetzt wird). Die Status werden wie folgt lauten:

	N	V		B	D	I	Z	C
C1 =	1	1	0	0	0	0	0	1

Beachten Sie, daß das Unterbrechungs-Maskenbit gesetzt oder gelöscht wird, abhängig von seinem Wert zu der Zeit, bei der das Statusregister gespeichert wurde, unter der Annahme, daß die Unterbrechungs-Serviceroutine es nicht ändert, während es sich auf dem Stapel befand.

RTS – RETURN FROM SUBROUTINE (KEHRE VON UNTERPROGRAMM ZURÜCK)

Dieser Befehl holt einen neuen Befehlszähler-Wert von der Spitze des Stapels und inkrementiert ihn, bevor er zum Holen eines Befehls verwendet wird. Beachten Sie, daß der Stapelzeiger vor dem Laden jedes Datenbytes inkrementiert wird und sein endgültiger Wert um 2 größer ist als sein Anfangswert. RTS wird normalerweise am Ende eines Unterprogramms zum Zurückspeichern der Rückkehr-Adresse verwendet, die im Stapel durch einen JSR-Befehl aufbewahrt wurde. Erinnern Sie sich daran, daß die Rückkehr-Adresse,die durch JSR aufbewahrt wurde, in Wirklichkeit die Adresse des dritten Bytes des JSR-Befehls selbst ist. Daher muß RTS diese Adresse inkrementieren, bevor sie zur Wiederaufnahme des Hauptprogramms verwendet wird. Der vorausgehende Inhalt des Befehlszählers geht verloren. Jedes Unterprogramm muß wenigstens einen RTS-Befehl enthalten.

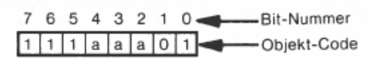


Durch einen RTS-Befehl werden keine Status geändert.

Der Stapelzeiger enthalte angenommen DF_{16}, der Speicherplatz $01E0_{16}$ enthalte 08_{16}, und der Speicherplatz $01E1_{16}$ enthalte $7C_{16}$. Nachdem der Befehl RTS ausgeführt wurde, wird im Stapelzeiger $E1_{16}$ und im Befehlszähler $7C09_{16}$ liegen (dies ist die Adresse, von wo ab die Befehls-Ausführung fortgesetzt wird).

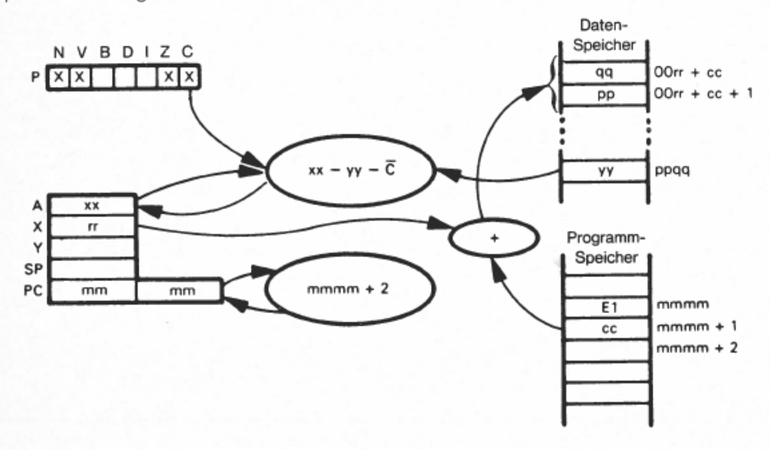
SBC – SUBTRACT MEMORY FROM ACCUMULTOR WITH BORROW (SUBTRAHIERE SPEICHER VOM AKKUMULATOR MIT BORGEN)

Subtrahiert den Inhalt des gewählten Speicherbytes und das Komplement des Übertrags-Status (d.h. 1 – C) vom Inhalt des Akkumulators. Dieser Befehl bietet die gleichen Speicher-Adressiermöglichkeiten wie der ADC-Befehl. Das erste Byte des Objektcodes wählt die Adressier-Art wie folgt:



Bit-Wert für aaa	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
000	E1	Indirekt, vor-indiziert mit X	2
001	E5	Null-Seite (direkt)	2
010	E9	Unmittelbar	2
011	ED	Absolut (direkt)	3
100	F1	Indirekt, nach-indiziert mit Y	2
101	F5	Null-Seite indiziert mit X	2
110	F9	Absolut indiziert mit Y	3
111	FD	Absolut indiziert mit X	3

Wir wollen den SBC-Befehl unter Verwendung vor-indizierter indirekter Adressierung (über das Indexregister X) illustrieren. Schlagen Sie die Besprechung der Adressier-Arten und anderer arithmetischer und logischer Befehle für Beispiele der übrigen Adressier-Arten nach.

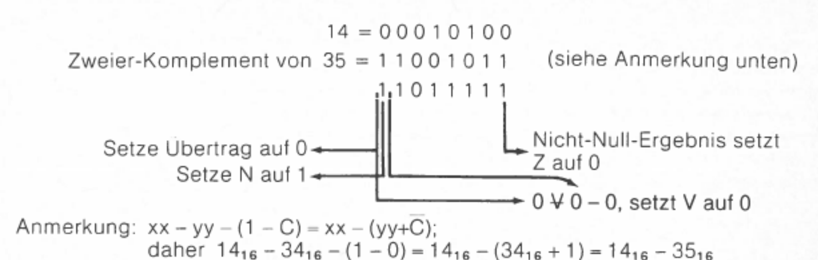


Subtrahiere den Inhalt des gewählten Speicherbytes und das Komplement des Übertrags-Status (1 – C) vom Akkumulator, wobei alle Register-Inhalte als einfache Binärdaten behandelt werden. Beachten Sie jedoch, daß alle Daten dezimal (BCD) behandelt werden, wenn der D-Status gesetzt ist.

Es sei angenommen xx = 14_{16}, cc = 15_{16}, rr = 37_{16}, ppqq = $07E2_{16}$, yy = $(07E2_{16}) = 34_{16}$ und C = 0. Nach Ausführung eines Befehls

SBC ($15,X)

würde der Inhalt des Akkumulators auf DF_{16} geändert.



Anmerkung: xx – yy – (1 – C) = xx – (yy+C̄);
daher $14_{16} - 34_{16} - (1 - 0) = 14_{16} - (34_{16} + 1) = 14_{16} - 35_{16}$

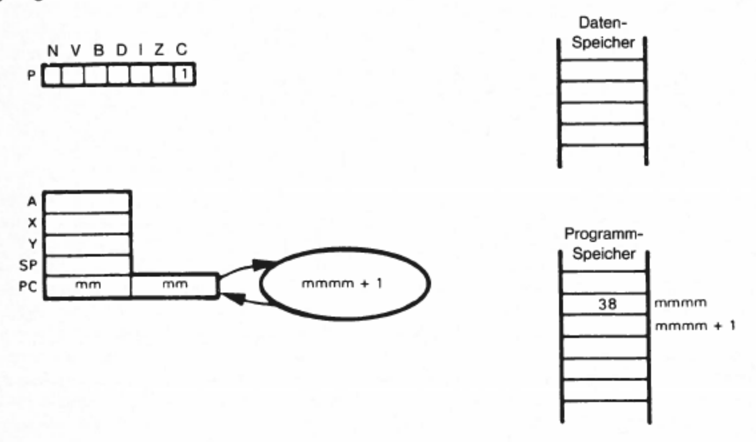
Beachten Sie, daß der resultierende Übertrag kein Borgen darstellt. Er ist vielmehr das Gegenteil eines Borgens, da er auf 1 gesetzt ist, wenn kein Borgen erforderlich ist, und gelöscht, wenn ein Borgen erforderlich ist. Sie sollten sehr sorgfältig diese Verwendung beachten, da sie sich von den meisten anderen Mikroprozessoren unterscheidet, die den Übertrag komplementieren, bevor er nach einer Subtraktion gespeichert wird.

SBC ist der einzige binäre Subtraktions-Befehl. Um ihn in Einzelbyte-Operationen zu verwenden oder um die niederwertigen Bytes zweier Multibyte-Zahlen zu subtrahieren, muß ein vorhergehender Befehl (SEC) ausdrücklich C auf 1 setzen, so daß dieser die Operation nicht beeinflußt. Erinnern Sie sich daran, daß C vor einer Subtraktion gesetzt (nicht gelöscht) werden muß, da seine Bedeutung entgegengesetzt dem gewöhnlichen Borgen ist. Beachten Sie auch, daß der Mikroprozessor 6502 – anders als die meisten anderen – keinen Subtraktions-Befehl besitzt, der den Übertrag nicht enthält.

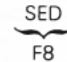
SEC – SET CARRY (SETZE ÜBERTRAG)



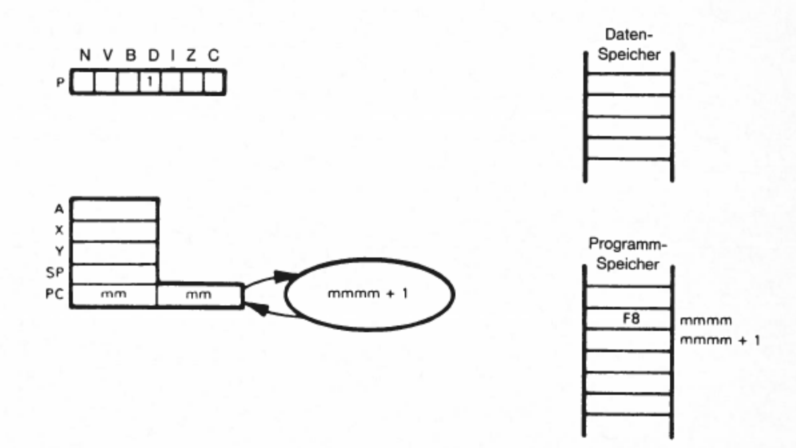
Setzt den Übertrags-Status auf 1. Es werden keine anderen Status oder Register-Inhalte beeinflußt. Beachten Sie, daß dieser Befehl als Teil einer normalen Subtraktions-Operation benötigt wird, da der einzige beim Mikroprozessor 6502 verfügbare Subtraktions-Befehl der SBC ist, der auch den komplementierten Übertrags-Status subtrahiert. Dieser Befehl ist auch beim Beginn einer Multibyte-Subtraktion erforderlich, da es niemals ein Borgen vom niedrigstwertigen Byte gibt.



SED – SET DECIMAL MODE (SETZE DEZIMAL-BETRIEBSART)



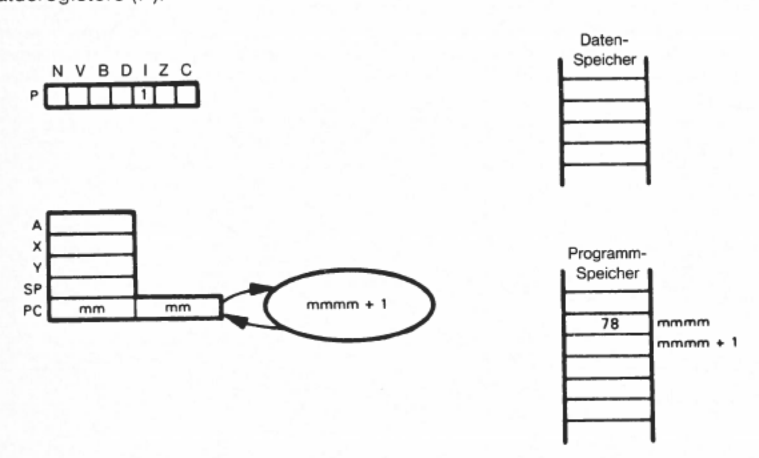
Setzt den Dezimal-Betriebsart-Status auf 1. Es wird kein anderer Status oder Register-Inhalt beeinflußt. Dieser Befehl wird dazu verwendet, um den Prozessor 6502 in die Dezimal-Betriebsart zu versetzen, in der ADC- und SBC-Befehle BCD-Ergebnisse anstatt binärer Resultate liefern. Der Programmierer sollte sehr sorgfältig auf die Tatsache achten, daß das gleiche Programm unterschiedliche Resultate erzeugen wird, abhängig vom Zustand des Dezimal-Betriebsarts-Status. Dies kann zu unangenehmen und überraschenden Fehlern führen, wenn der Zustand des Dezimal-Betriebsarts-Status nicht sorgfältig beobachtet wird.



SEI – SET INTERRUPT MASK (DISABLE INTERRUPT) (SETZE UNTERBRECHUNGS-MASKE – SPERRE UNTERBRECHUNG)

SEI
78

Setzt die Unterbrechungs-Maske im Statusregister. Dieser Befehl sperrt die Unterbrechungs-Servicemöglichkeit des 6502, das heißt, der 6502 wird auf die Unterbrechungs-Anforderungs-Steuerleitung nicht reagieren. Es werden keine anderen Register oder Status beeinflußt. Die Unterbrechungs-Maske ist Bit 2 des Statusregisters (P).



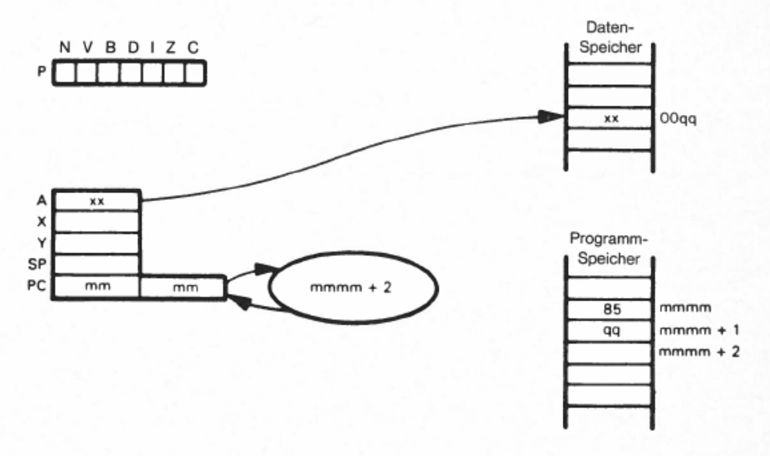
STA – STORE ACCUMULATOR IN MEMORY (SPEICHERE AKKUMULATOR IN SPEICHER)

Speichert den Inhalt des Akkumulators in den spezifizierten Speicherplatz. Diese Befehl bietet die gleichen Speicher-Adressiermöglichkeiten wie der ADC-Befehl mit der Ausnahme, daß eine unmittelbare Adressier-Möglichkeit nicht vorhanden ist. Das erst Byte des Objektcodes wählt die Adressier-Art wie folgt:



Bit-Wert für aaa	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
000	81	Indirekt, vor-indiziert mit X	2
001	85	Null-Seite (direkt)	2
010	89	Nicht verwendet	
011	8D	Absolut (direkt)	3
100	91	Indirekt, nach-indiziert mit Y	2
101	95	Null-Seite indiziert mit X	2
110	99	Absolut indiziert mit Y	3
111	9D	Absolut indiziert mit X	3

Wir wollen den STA-Befehl mit direkter Nullseiten-Adressierung illustrieren. Schlagen Sie die Besprechung der Adressier-Arten und anderer arithmetischer und logischer Befehle für Beispiele der übrigen Adressier-Arten nach. Es werden keine Status beeinflußt.



Speichere den Inhalt des Akkumulators in den Speicher. Es sei angenommen xx = 63_{16} und qq = $3A_{16}$. Nachdem der Befehl

STA $3A

ausgeführt wurde, wird der Inhalt des Speicherplatzes $003A_{16}$ gleich 63_{16} sein. Es werden keine Register oder Status beeinflußt.

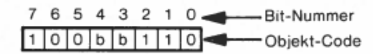
STX – STORE INDEX REGISTER X IN MEMORY (SPEICHERE INDEXREGISTER X IN SPEICHER)

Speichert den Inhalt des Indexregisters X in den gewählten Speicherplatz. Die zulässigen Adressier-Arten sind:

1) Null-Seite (direkt) - STX addr
2) Absolut (direkt) - STX addr16
3) Null-Seite indiziert mit Y - STX addr,Y

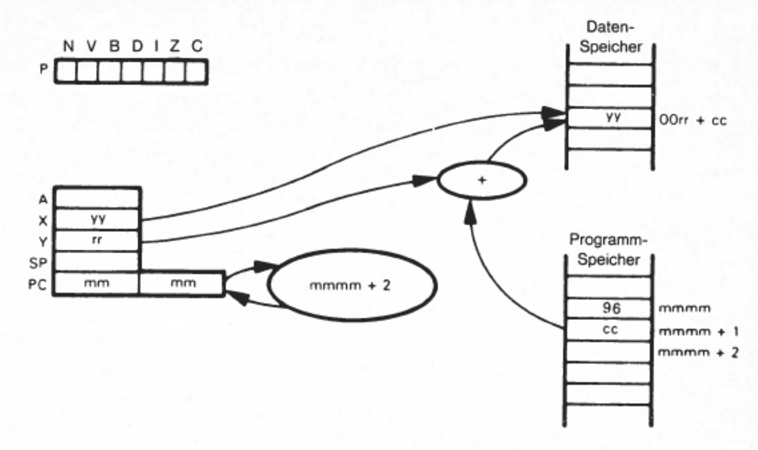
Beachten Sie, daß es hier keine indizierten Adressier-Arten unter Verwendung des Indexregisters X gibt. Es gibt auch keine absolute indizierte Adressierung. STX und LDX sind die einzigen Befehle, die die indizierte Nullseiten-Adressierung mit Indexregister Y verwenden. Es werden keine Status beeinflußt.

Das erste Byte des Objektcodes wählt die Adressier-Art wie folgt:



Bit-Wert für aaa	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
00	86	Null-Seite (direkt)	2
01	8E	Absolut (direkt)	3
10	96	Null-Seite indiziert mit Y	2
11	9E	Nicht verwendet	

Wir wollen den STX-Befehl unter Verwendung der indizierten Nullseiten-Adressierung mit Indexregister Y zeigen. Schlagen Sie die Besprechung der Adressier-Arten und anderer arithmetischer und logischer Befehle für Beispiele der übrigen Adressier-Arten nach.



Speichere den Inhalt des Indexregisters X in das gewählte Speicherbyte. Es sei angenommen cc $=28_{16}$, rr $= 20_{16}$ und yy $= E9_{16}$. Nach Ausführung des Befehls

STX $28,Y

wird der Speicherplatz 0048_{16} den Wert $E9_{16}$ enthalten. Es werden keine Register oder Status beeinflußt.

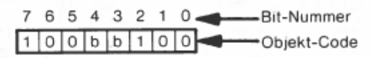
STY – STORE INDEX REGISTER Y IN MEMORY (SPEICHERE INDEXREGISTER Y IN DEN SPEICHER)

Speichert den Inhalt des Indexregisters Y in den gewählten Speicherplatz. Die zulässigen Adressier-Arten sind:

1) Null-Seite (direkt) - STY addr
2) Absolut (direkt) - STY addr 16
3) Null-Seite indiziert mit X - STY addr,X

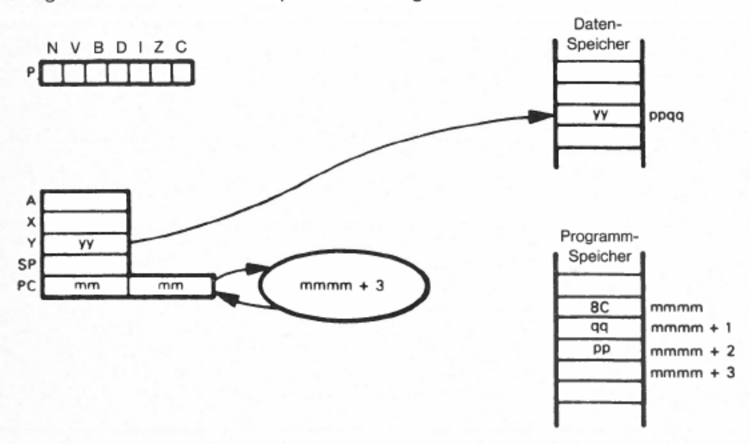
Beachten Sie, daß keine indizierten Adressier-Arten unter Verwendung des Indexregisters Y gibt. Es gibt auch keine absolute indizierte Adressier-Art. Es werden keine Status oder Register beeinflußt.

Das erste Byte des Objektcodes wählt die Adressier-Art wie folgt:



Bit-Wert für aaa	Hexadezimaler Objekt-Code	Adressier-Art	Anzahl der Bytes
00	84	Null-Seite (direkt)	2
01	8C	Absolut (direkt)	3
10	94	Null-Seite indiziert mit X	2
11	9C	Nicht verwendet	

Wir wollen den STY-Befehl mit absoluter direkter Adressierung illustrieren. Schlagen Sie die Besprechung der Adressier-Arten und anderer arithmetischer und logischer Befehle für Beispiele der übrigen Adressier-Arten nach.



Speichere den Inhalt des Indexregisters Y in das gewählte Speicherbyte. Es sei angenommen yy = 01_{16} und ppqq = $08F3_{16}$. Nachdem der Befehl

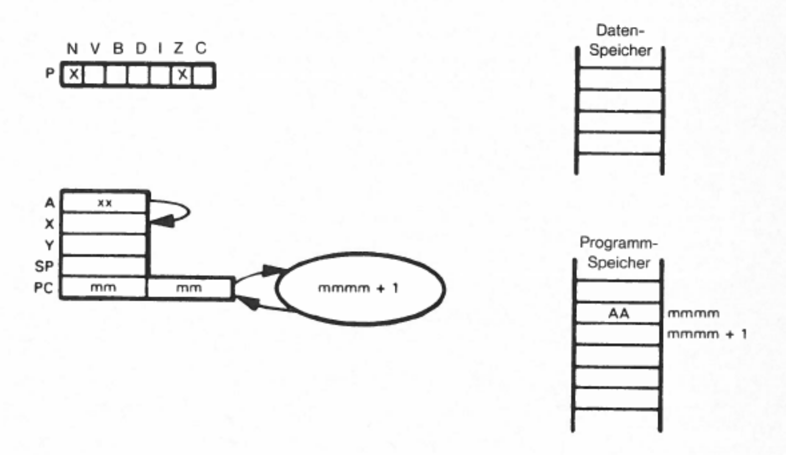
STY $08F3

ausgeführt wurde, wird der Speicherplatz $08F3_{16}$ den Wert 01_{16} enthalten. Es werden keine Register oder Status beeinflußt.

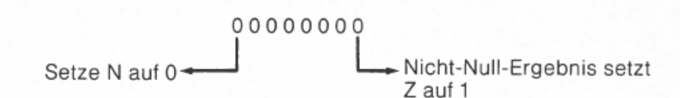
TAX – MOVE FROM ACCUMULATOR TO INDEX REGISTER X (TRANSFERIERE VOM AKKUMULATOR ZUM INDEXREGISTER X)

TAX
AA

Bringt den Inhalt des Akkumulators zum Indexregister X. Setzt den Negativ- und Null-Status entsprechend.



Es sei angenommen xx = 00_{16}. Nach Ausführung des TAX-Befehls wird sowohl der Akkumulator wie das Indexregister X den Wert 00_{16} enthalten.



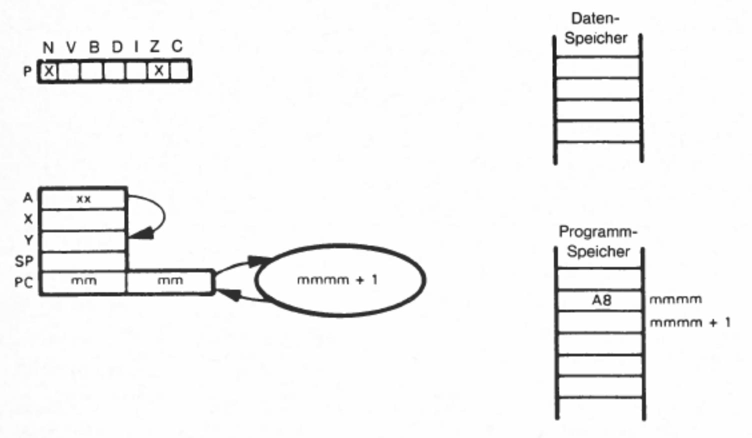
Die folgende Befehls-Sequenz wird den Inhalt des Indexregisters vom Stapel nach Abschluß eines Unterprogrammes oder einer Unterbrechungs-Service-Routine zurückspeichern:

```
PLA   ;HOLE ALTES X-REGISTER VOM STAPEL
TAX   ;SPEICHERE ZUM X-REGISTER ZURÜCK
```

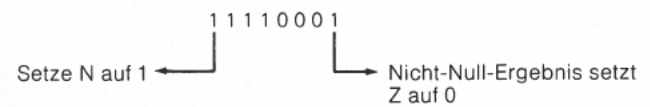
TAY – MOVE FROM ACCUMULATOR TO INDEX REGISTER Y (TRANSFERIERE VOM AKKUMULATOR ZUM INDEXREGISTER Y)

TAY
A8

Bringt den Inhalt des Akkumulators zum Indexregister Y. Setzt den Negativ- und Nullstatus entsprechend.



Es sei angenommen xx = $F1_{16}$. Nach Ausführung des TAY-Befehls werden sowohl der Akkumulator wie das Indexregister Y den Wert $F1_{16}$ enthalten.



Die folgende Befehls-Sequenz wird den Inhalt des Indexregisters Y vom Stapel nach Abschluß eines Unterprogrammes oder einer Unterbrechungs-Sevice-Routine zurückspeichern:

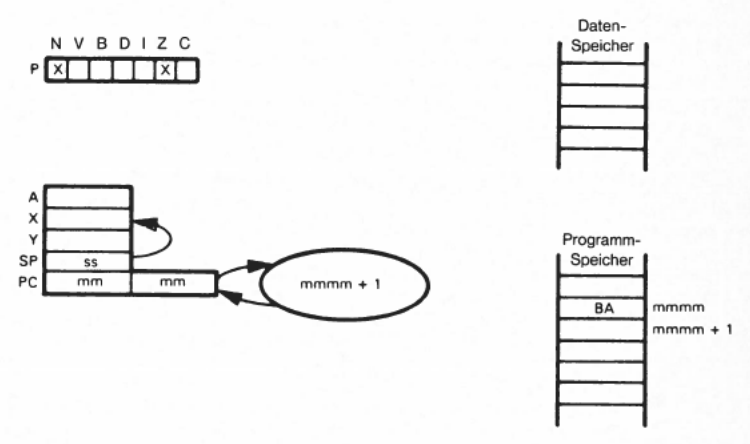
```
PLA   ;HOLE ALTES Y-REGISTER VOM STAPEL
TAY   ;SPEICHERE ES ZUM Y-REGISTER ZURÜCK
```


TSX – MOVE FROM STACK POINTER TO INDEX REGISTER X (TRANSFERIERE VOM STAPELZEIGER ZUM INDEXREGISTER X)

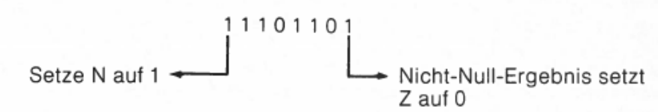
TSX
BA

Bringt den Inhalt des Stapelzeigers zum Indexregister X. Setzt den Negativ- und Nullstatus entsprechend. Beachten Die, daß TSX der einzige Befehl des 6502 ist, der Ihnen den Zugriff zum Stapelzeiger gestattet. Eine typische Befehls-Sequenz, die den Wert des Stapelzeigers im Speicherplatz TEMP aufbewahrt ist:

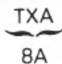
```
TSX          ;BRINGE STAPELZEIGER ZU X
STX   TEMP   ;BEWAHRE STAPELZEIGER IM SPEICHER AUF
```



Wenn beispielsweise der Stapelzeiger ED_{16} nach Ausführung des TSX-Befehls den Wert ED_{16} enthält, werden sowohl der Stapelzeiger wie das Indexregister X den Wert ED_{16} enthalten.

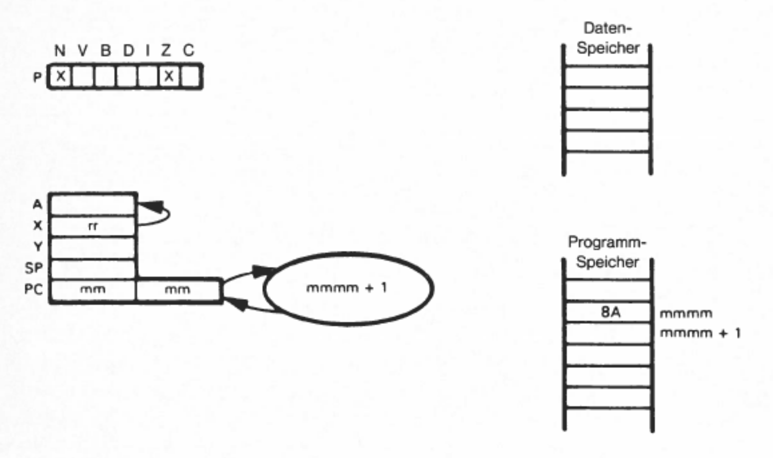


TXA – MOVE FROM INDEX REGISTER X TO ACCUMULATOR (TRANSFERIERE VOM INDEXREGISTER X ZUM AKKUMULATOR)

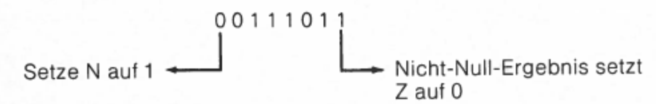


Bringt den Inhalt des Indexregisters X zum Akkumulator und setzt den Negativ- und Nullstatus entsprechend. Die folgende Befehls-Sequenz wird den Inhalt des Indexregisters X im Stapel vor der Ausführung eines Unterprogrammes oder einer Unterbrechungs-Service-Routine aufbewahren:

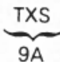
```
TXA  ;BRINGE X-REGISTER ZUM AKKUMULATOR
PHA  ;BEWAHRE X-REGISTER IM STAPEL AUF
```



Es sei angenommen rr = 38_{16}. Nach Ausführung des TXA-Befehls werden sowohl das Indexregister X wie der Akkumulator den Wert 38_{16} enthalten.



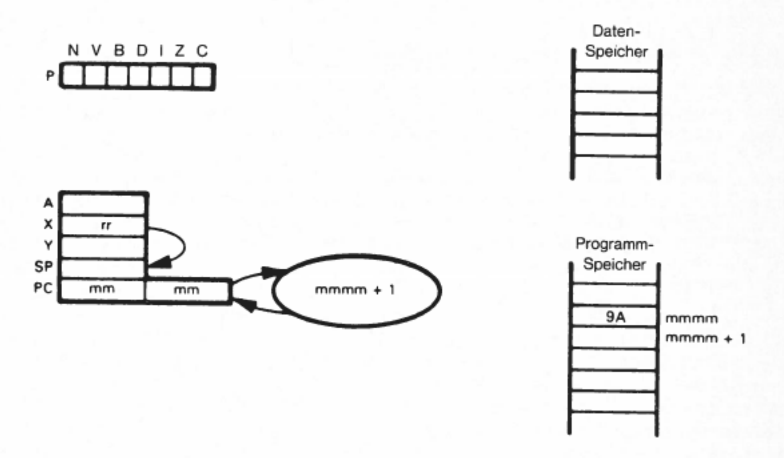
TXS – MOVE FROM INDEX REGISTER X TO STACK POINTER (TRANSFERIERE VOM INDEXREGISTER X ZUM STAPELZEIGER)



Bringt den Inhalt des Indexregisters X zum Stapelzeiger. Es werden keine anderen Register oder Status beeinflußt. Beachten Sie, daß TXS der einzige Befehl des 6502 ist, der Ihnen die Bestimmung des Wertes im Stapelzeiger gestattet. Eine typische Befehls-Sequenz, die den Stapelzeiger mit dem Wert LAST lädt lautet:

```
LDX  #LAST   ;HOLE LAGE DES STAPELS AUF SEITE 1
TXS          ;PLAZIERE START-PLATZ IN STAPELZEIGER
```

Beachten Sie, daß TXS keinen Status beeinflußt, im Gegensatz zu TSX, der sowohl den Null- wie den Negativ-Status beeinflußt.



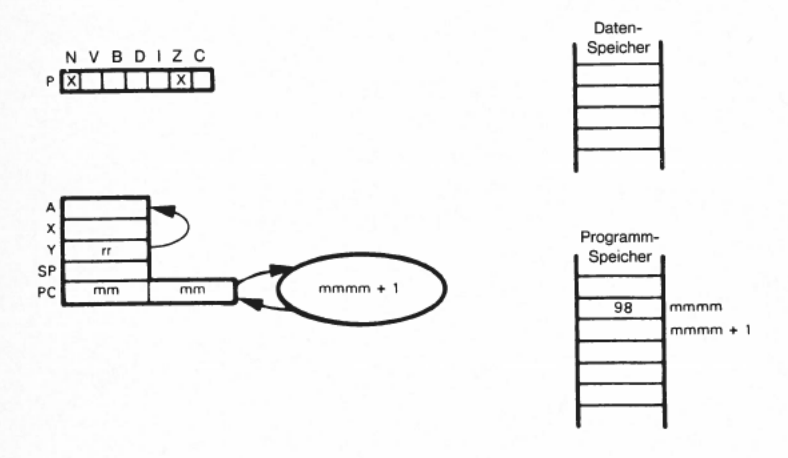
Es sei angenommen rr = $F2_{16}$. Nach Ausführung des TXS-Befehls enthalten sowohl das Indexregister wie der Stapelzeiger den Wert $F2_{16}$, wodurch $01F2_{16}$ zum momentanen Ort des Stapels wird. Es werden keine Status oder andere Register beeinflußt.

TYA – MOVE FROM INDEX REGISTER Y TO ACCUMULATOR (TRANSFERIERE VOM INDEXREGISTER Y ZUM AKKUMULATOR)

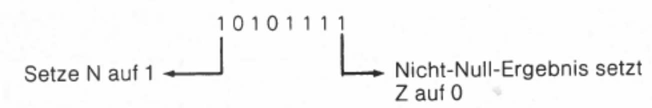
TYA
98

Bringt den Inhalt des Indexregisters Y zum Akkumulator und setzt den Negativ- und Nullstatus entsprechend. Die folgende Befehls-Sequenz wird den Inhalt des Indexregisters Y in den Stapel vor Ausführung eines Unterprogrammes oder einer Unterbrechungs-Service-Routine aufbewahren:

```
TYA  ;BRINGE Y-REGISTER ZUM AKKUMULATOR
PHA  ;BEWAHRE Y-REGISTER IM STAPEL AUF
```



Es sei angenommen rr = AF_{16}. Nach Ausführung des TYA-Befehls werden sowohl das Indexregister Y wie der Akkumulator den Wert AF_{16} enthalten.



KOMPATIBILITÄT 6800/6502

6800/6502-ÄHNLICHKEIT

Obwohl der Mikroprozessor 6502 an und für sich allein verwendet werden kann, ist eine seiner wesentlichen Eigenschaften seine Ähnlichkeit mit dem weit verbreiteten Mikroprozessor 6800. Diese Ähnlichkeit reicht jedoch nicht aus, um für diesen Prozessoren geschriebene Programme auf der Maschinen- oder Assembler-Ebene des anderen laufen zu lassen, reicht jedoch aus, daß Programmierer leicht von einer CPU zur anderen wechseln können. Die meisten der externen Hilfsbausteine, die für einen dieser Prozessoren entwickelt wurden, können auch mit dem anderen verwendet werden. Die Kapitel 9 und 10 von "An Introduction to Microcomputers: Volume 2 – Some Real Microprocessors" besprechen diese Hardware-Kompatibilität im Detail.

Wir wollen die Mikroprozessoren 6800 und 6502 kurz beschreiben und sie in Hinblick auf ihre Register, Status, Adressier-Arten und Befehlssätze vergleichen. Sie sollten beachten, daß keiner der beiden Prozessoren das Spiegelbild des anderen darstellt, sie besitzen jedoch weit mehr Ähnlichkeiten als gegenüber einem 8080, Z80, F8 oder 2650. Diese Beschreibung sollte Ihnen eine gewisse Vorstellung geben, welchen Problemen Sie gegenüberstehen, wenn Sie von einer CPU zu einer anderen überwechseln wollen.

6800/6502-REGISTER-VERGLEICH

Was die Register betrifft, besitzen sowohl der 6800 wie der 6502 einen primären Akkumulator (A-Register) mit 8-Bit und einen 16-Bit-Befehlszähler (oder PC-Register). Die übrigen Register jedoch sind etwas unterschiedlich. Der 6800 besitzt einen zweiten 8-Bit-Akkumulator (B-Register), ein 16-Bit-Indexregister und einen 16-Bit-Stapelzeiger. Der 6502 andererseits besitzt zwei 8-Bit-Indexregister und einen 8-Bit-Stapelzeiger. Daher können die Indexregister des 6502 nicht eine vollständige 16-Bit-Speicher-Adresse aufbewahren, während das Indexregister des 6800 dies kann. Ferner kann der RAM-Stapel des 6800 infolge seines 16-Bit-Stapelzeigers überall im Stapel liegen, während der RAM-Stapel des 6502 immer auf Seite 1 liegt.

6800/6502-STATUS-VERGLEICH

Für die Status gilt, daß der 6800 und der 6502 identische Null-, Überlauf-, Negativ- und Unterbrechungsmasken-Status besitzen. Der Unterschied beim Übertrags-Status liegt darin, daß die Version dieses Flags beim 6800 und 6502 entgegengesetzte Bedeutung nach Subtraktions-Operationen haben, Der Übertrag des 6800 wird auf 1 gesetzt, wenn ein Borgen erforderlich ist und andernfalls auf 0. Der Übertrag des 6502 wird auf 0 gesetzt, wenn ein Borgen erforderlich ist, und andernfalls auf 1. Dieser Unterschied bedeutet, daß vor einer Multibyte-Subtraktions-Operationen der Programmierer den Übertrag beim 6800 löschen und den Übertrag beim 6502 setzen muß. **Der 6800 und 6502 unterscheiden sich auch darin, wie sie Dezimal-Arithmetik ausführen.** Der 6800 besitzt ein Halb-Übertrags-Flag (oder Übertrag von Bit 3), während der 6502 ein Flag für die Dezimal-Betriebsart besitzt. **Ferner hat der 6502 ein Break-Flag, das es beim 6800 nicht gibt.** Es ist beim 6800 nicht erforderlich, da der Software-Unterbrechungsbefehl des 6800 automatisch vektorisiert wird, unabhängig von der normalen Unterbrechungs-Reaktion.

6800/6502-ADRESSIER-ART VERGLEICH

Der Mikroprozessor 6502 besitzt wesentlich mehr Adressier-Arten als der 6800. Dies ist besonders deshalb erforderlich, da die Indexregister des 6502 nur 8-Bits lang sind. Tabelle 3-7 vergleicht die bei den beiden Prozessoren verfügbaren Adressier-Arten. Der Mikroprozessor 6800 besitzt keine indirekte Adressier-Art, keine Kombinationen von Indizierung und indirektem Vorgehen und keine absoluten indizierten Adressier-Arten. Es gibt auch einige andere Unterschiede bei den Ausdrücken, welche Betriebsarten mit bestimmten Befehlen verfügbar sind. Wir wollen diese Unterschiede nicht besprechen, sie sind jedoch in Tabelle 3-6 aufgezählt.

Tabelle 3-7. Bei den Mikroprozessoren 6800 und 6502 verfügbare Speicheradressierarten.

6800	6502
Unmittelbar	Unmittelbar
Direkt (Null-Seite)	Null-Seite (direkt)
Erweitert (absolut direkt)	Absolut (direkt)
Indiziert (absolut)	Absolut indiziert
	Null-Seite indiziert
	Nach-indiziert indirekt
	Vor-indiziert indirekt
	Indirekt
Relativ (nur Verzweigungen)	Relativ (nur Verzweigungen)

Beachten Sie, daß zahlreiche unterschiedliche Varianten der indizierten Adressierung beim Mikroprozessor 6502 verfügbar sind, aber erinnern Sie sich daran, daß die Index-Register des 6502 nur 8 Bit lang sind, während das Indexregister des 6800 eine Länge von 16 Bit besitzt.

6800/6502 VERGLEICH DER BEFEHLE

Die Befehlssätze des 6800 und 6502 sind ähnlich, jedoch nicht identisch (siehe Tabelle 3-6). Tabelle 3-8 vergleicht diese beiden Sätze, wobei zuerst die Befehle aufgelistet sind, die in beiden vorliegen, dann die Befehle des 6800, die kein Äquivalent beim 6502 besitzen, und schließen die Befehle des 6502, die kein Äquivalent beim 6800 besitzen. Offensichtlich resultieren diese Unterschiede aus der Verschiedenheit der Status und Register. Zahlreiche dieser Unterschiede sind geringfügig, und schließen Befehle ein, die ein kleiner Teil der allgemeinen Anwendungsprogramme sind. Ein offensichtlicher Unterschied besteht darin, daß der 6800 Additions- und Subtraktions-Befehle besitzt, die den Übertrags-Status (ADD und SUB) nicht beinhalten, während dies beim 6502 nicht der Fall ist. Dies bedeutet, daß der Assemblersprachen-Programmierer beim 6502 ausdrücklich den Übertrags-Status löschen oder setzen muß, wenn dessen Wert eine Addition-oder Subtraktions-Operation nicht beeinflussen soll. Beachten Sie, daß diese Ähnlichkeit der Befehlssätze nicht mehr für die Objektcode-Ebene gilt. Die tatsächlichen Maschinencodes sind bei den beiden Mikroprozessoren völlig verschieden.

Tabelle 3-8. Vergleich der Assemblersprache-Befehlssätze des 6800 und 6502.

I.	Gemeinsame Befehle
Befehl	Bedeutung
ADC	Add with Carry (Addiere mit Übertrag)
AND	Logical AND (UNDIere logisch)
ASL	Arithmetic Shift Left (Schiebe arithmetisch nach links)
BCC	Branch if Carry Clear (Verzweige, wenn Übertrag gelöscht)
BCS	Branch if Carry Set (Verzweige, wenn Übertrag gesetzt)
BEQ	Branch if Equal to Zero (Z = 1) (Verzweige, wenn gleich Null (Z = 1))
BIT	Bit Test (Teste Bit)
BMI	Branch if Minus (N = 1) (Verzweige, wenn minus (N = 1))
BNE	Branch if Not Equal to Zero (Z = 0) (Verzweige, wenn nicht gleich Null (Z = 0))
BPL	Branch If Plus (N = 0) (Verzweige, wenn plus (N = 0))
BVC	Branch if Overflow Clear (Verzweige, wenn Überlauf gelöscht)
BVS	Branch if Overflow Set (Verzweige, wenn Überlauf gesetzt)
CLC	Clear Carry (Lösche Übertrag)
CLI	Clear Interrupt Mask (Enable Interrupt) (Lösche Unterbrechungs-Maske (Gib Unterbrechung frei))
CLV	Clear Overflow (Lösche Überlauf)
CMP	Compare Accumulator with Memory (Vergleiche Akkumulator mit Speicher)
CPX[1] (auch CPY beim 6502)	Compare Index Register with Memory (Vergleiche Index-Register mit Speicher)
DEC	Decrement (by 1) (Dekrementiere um 1)
DEX[1] (auch DEY beim 6502)	Decrement Index Register (by 1) (Dekrementiere Index-Register um 1)
EOR	Logical Exclusive-OR (Exklusiv-ODERiere logisch)
INC	Increment (by 1) (Inkrementiere um 1)
INX[1] (auch INY beim 6502)	Increment Index Register (by 1) (Inkrementiere Index-Register um 1)
JMP	Jump to New Location (Springe zu neuem Speicherplatz)
JSR	Jump to Subroutine (Springe zu Unterprogramm)
LDA	Load Accumulator (Lade Akkumulator)
LDX[1] (auch LDY beim 6502)	Load Index Register (Lade Index-Register)
LSR	Logical Shift Right (Verschiebe logisch nach links)
NOP	No Operation (Keine Operation)
ORA	Logical (Inclusive) OR (ODERiere logisch)
PHA (PSH beim 6800)	Push Accumulator onto Stack (Bringe Akkumulator zu Stapel)
PLA (PUL beim 6800)	Pull Accumulator from Stack (Hole Akkumulator vom Stapel)
ROL	Rotate Left through Carry (Rotiere nach links durch Übertrag)
ROR	Rotate Right through Carry (Rotiere nach rechts durch Übertrag)
RTI	Return from Interrupt (Kehre von Unterbrechung zurück)
RTS	Return from Subroutine (Kehre von Unterprogramm zurück)
SBC[2]	Subtract with Carry (Subtrahiere mit Übertrag)
SEC	Set Carry (Setze Übertrag)
SEI	Set Interrupt Mask (Setze Unterbrechungs-Maske)
STA	Store Accumulator (Speichere Akkumulator)
STX[1] (auch STY beim 6502)	Store Index Register (Speichere Index-Register)
TSX	Transfer Stack Pointer to Index Register (X) (Transferiere Stapelzeiger zu Index-Register (X))
TXS	Transfer Index Register (X) to Stack Pointer (Transferiere Index-Register (X) zu Stapelzeiger

[1]Das Index-Register X ist beim 6800 16 Bits lang, beim 6502 8 Bits lang, der auch ein Index-Register Y besitzt.

[2]Beachten Sie, daß SBC beim 6502 eine andere Bedeutung als beim 6800 hat, da bei Subtraktions-Operationen der Übertrag des 6800 das Gegenteil des Übertrags des 6502 ist.

Tabelle 3-8. Vergleich der Assemblersprache-Befehlssätze des 6800 und 6502 (Fortsetzung)

II.	Befehle nur für 6800
Befehl	**Bedeutung**
ABA	Add Accumulators (Addiere Akkumulatoren)
ADD	Add (without Carry) (Addiere (ohne Übertrag))
ASR	Arithmetic Shift Right (Verschiebe arithmetisch nach links)
BGE	Branch if Greater than or Equal to Zero (Verzweige, wenn größer oder gleich Null)
BGT	Branch if Greater than Zero (Verzweige, wenn größer als Null)
BHI	Branch if Higher (Verzweige, wenn höher)
BLE	Branch if Less than or Equal to Zero (Verzweige, wenn kleiner oder gleich Null)
BLS	Branch if Lower or Same (Verzweige, wenn kleiner oder gleich)
BLT	Branch if Less than Zero (Verzweige, wenn kleiner als Null)
BRA	Branch Unconditionally (Verzweige unbedingt)
BSR	Branch to Subroutine (Verzweige zu Unterprogramm)
CBA	Compare Accumulators (Vergleiche Akkumulatoren)
CLR	Clear (Lösche)
COM	Logical Complement (Komplementiere logisch)
DAA	Decimal Adjust Accumulator (Ordne Akkumulator dezimal an)
DES	Decrement Stack Pointer (by 1) (Dekrementiere Stapelzeiger um 1)
INS	Increment Stack Pointer (by 1) (Inkrementiere Stapelzeiger um 1)
LDS	Load Stack Pointer (Lade Stapelzeiger)
NEG	Negate (Twos Complement) (Negiere (Zweier-Komplement))
SBA	Subtract Accumulators (Subtrahiere Akkumulatoren)
SEV	Set Overflow (Setze Überlauf)
STS	Store Stack Pointer (Speichere Stapelzeiger)
SUB	Subtract (without Carry) (Subtrahiere (ohne Übertrag))
SWI	Software Interrupt (like 6502 BRK) (Software-Unterbrechung (wie 6502-Break))
TAB	Move from Accumulator A to Accumulator B (Bringe von Akkumulator A zu Akkumulator B)
TAP	Move from Accumulator A to CCR (Bringe von Akkumulator A zu CCR)
TBA	Move from Accumulator B to Accumulator A (Bringe von Akkumulator B zu Akkumulator A)
TPA	Move CCR to Accumulator A (Bringe CCR zu Akkumulator A)
TST	Test Zero or Minus (Teste auf Null oder minus)
WAI	Wait for Interrupt (Warte auf Unterbrechung)

III.	Befehle nur für 6502
Befehl	**Bedeutung**
BRK	Break (like 6800 SWI) (Break (wie 6800 SWI))
CLD	Clear Decimal Mode (Lösche Dezimal-Betriebsart)
PHP	Push Status Register onto Stack (Bringe Status-Register zum Stapel)
PLP	Pull Status Register from Stack (Hole Status-Register vom Stapel)
SED	Set Decimal Mode (Setze Dezimal-Betriebsart)
TAX (TAY)	Transfer Accumulator to Index Register X (Y) (Transferiere Akkumulator zu Index-Register X (Y))
TXA (TYA)	Transfer Index Register X (Y) to Accumulator (Transferiere Index-Register X (Y) zu Akkumulator)

ASSEMBLER-VEREINBARUNGEN FÜR DEN MOS-TECHNOLOGY 6502

Der Standard-Assembler ist von den Herstellern des 6502 erhältlich sowie auf den wichtigsten Time-Sharing-Netzwerken verfügbar. Er ist auch in den meisten Entwicklungs-Systemen enthalten. Cross-Assmbler-Versionen sind für die meisten großen Computer und für zahlreiche Minicomputer erhältlich.

FELD-AUFBAU DES ASSEMBLERS

Die Befehle in Assemblersprache haben den Standard-Feldaufbau (siehe Tabelle 2-1). Die erforderlichen Begrenzungzeichen sind:

1) **Ein Zwischenraum nach einer Marke. Beachten Sie, daß alle Marken in Spalte 1 beginnen müssen.**
2) **Ein Zwischneraum nach dem Operationscode.**
3) **Ein Komma zwischne Operanden im Adressen-Feld, d.h., zwischen der Versetzungs-Adresse und X oder Y zur Anzeige der Indizierung mit Indexregister X oder Y.**
4) **Klammern um Adressen, die indirekt zu verwenden sind.**
5) **Ein Strichpunkt oder Rufzeichen (wir wollen den Strichpunkt verwenden) vor einem Kommentar.**

Typische Assemblersprachen-Befehle des 6502 sind:

```
START  LDA  (1000,X)     ;HOLE LÄNGE
       ADC  NEXT
LAST   BRK               ;ENDE DES ABSCHNITTES
```

MARKIERUNGEN (LABELS)

Der Assembler gestattet oft nur sechs Zeichen in Markierungen und unterdrückt weitere. Das erste Zeichen muß ein Buchstabe sein, während darauffolgende Zeichen Buchstaben oder Ziffern sein müssen. Die Einzelzeichen A, X und Y sind für den Akkumulator und die beiden Indexregister reserviert. Die Verwendung von Operationscodes als Markierungen ist häufig nicht gestattet und stellt auch keine gute Programmier-Technik dar.

PSEUDO-OPERATIONEN

Der Assembler besitzt die folgenden bestimmten Pseudo-Operationen:

.BYTE – Form Byte-Lenght Data
.DBYTE – Form Double-Byte-Lenght Data with MSBs First
.END – End of Program
.TEXT – Form String of ASCII Charakters
.WORD – Form Double-Byte-Lenght Data with LSBs First
= – Equata

Andere Pseudo-Operationen können ausgeführt werden, indem der Stellenzähler des Assemblers (gekennzeichnet mit *) auf einen neuen Wert gebracht wird. Beispiele sind:

*** = ADDR – Setze Programm-Anfang auf ADDR**
*** = *+N – Reserviere N Bytes für Datenspeicher**

.BYTE, .DBYTE, .TEXT, .WORD PSEUDO-OPERATIONEN

.BYTE, .DBYTE, .TEXT und .WORD sind die Daten-Pseudo-Operationen, die zum Plazieren von Daten in ein ROM verwendet werden.
.BYTE wird für 8-Bit-Daten verwendet, .TEXT für 7-Bit-ASCII-Zeichen (MSB ist null), .DBYTE für 16-Bit-Daten mit den höchstwertigen Bits zuerst und .WORD für 16-Bit-Adressen oder Daten mit dem niedrigstwertigen Bit zuerst. Beachten Sie speziell den Unterschied zwischen .DBYTE und .WORD.

Beispiele:

ADDR .WORD $3165

ergibt (ADDR) = 65 und (ADDR+1) = 31 (hex).

TCONV .BYTE 32

Diese Pseudo-Operation plaziert die Zahl 32 (20_{16}) in das nächste Byte des ROMs und weist den Namen TCONV der Adresse dieses Bytes zu.

ERROR .TEXT /ERROR/

Diese Pseudo-Operation plaziert die 7-Bit-ASCII-Zeichen E, R, R, O und R in die nächsten fünf Bytes des ROMs und weist den Namen ERROR der Adresse des ersten Bytes zu. Jedes Einzelzeichen (nicht nur /) kann zum Einrahmen des ASCII-Textes verwendet werden, aber wir wollen immer den / in Hinkunft verwenden.

MASK .DBYTE $1000

ergibt (MASK) = 10 und (MASK +1) = 00.

OPERS .WORD FADD, FSUB, FMUL, FDIV

Diese Pseudo-Operation plaziert die Adressen FADD, FSUB, FMUL und FDIV in die nächsten acht Speicherbytes (niedrigstwertige Bytes zuerst) und weist den Namen OPERS der Adresse des ersten Bytes zu.

SET ORIGIN-PSEUDO-OPERATION

Die Operation *=* +N ist die Reserve-Pseudo-Operation, die zum Zuweisen von Speicherplätzen im RAM verwendet wird.
Sie weist eine gegebene Anzahl von Bytes zu. *=* ist die Equate- oder die Define-Pseudo-Operation, die zur Definition von Namen verwendet wird. *=ADDR ist die Standard-Origin-Pseudo-Operation.

Programme für den 6502 besitzen gewöhnlich mehrere Origins, die wie folgt verwendet werden:

1) Zum Spezifieren der Reset- und Unterbrechungs-Service-Adresse. Diese Adressen müssen in die höchstwertigen Speicher-Adressen in dem System plaziert werden (gewöhnlich $FFFA_{16}$ bis $FFFF_{16}$).
2) Zum Spezifizieren der Start-Adressen der momentanen Reset- und Unterbrechungs-Service-Routinen. Diese Routinen selbst können irgendwo im Speicher plaziert werden.
3) Zum Spezifizieren der Start-Adresse des Hauptprogramms.
4) Zum Spezifizieren der Start-Adressen von Unterprogrammen.
5) Zur Definition von RAM-Speicherbereichen.
6) Zur Definition eines Bereiches (immer auf Seite 1) für den RAM-Stapel.
7) Zum Spezifizieren von Adressen, die für E/A-Ports und spezielle Funktionen dienen.

Beispiele:

```
RESET   =$3800
        *=$FFFC
        .WORD     RESET
        *=RESET
```

Anmerkung: $ bedeutet "hexadezimal"

Diese Sequenz plaziert die Reset-Befehls-Sequenz in den Speicher, beginnend bei Adresse 3800_{16} und plaziert diese Adresse in die Speicherplätze (Adressen $FFFC_{16}$ und $FFFD_{16}$), von wo die CPU des 6502 die Reset-Adressen holt.

Die hierauf folgende Befehls-Sequenz wird im Speicher aufbewahrt, beginnend beim Speicherplatz $C000_{16}$.

```
MAIN    =$C000
        *=MAIN
```

.END markiert einfach das Ende des Assemblersprachen-Programmes.

MARKIERUNGEN BEI PSEUDO-OPERATIONEN

Die Regeln und Empfehlungen für Markierungen bei Pseudo-Operationen des 6502 sind folgende:

1) Einfache Equates, so wie etwa MAIN = $C000, erfordern Markierungen, da ihr Zweck im Definieren der Bedeutung derartiger Markierungen besteht.
2) ,BYTE, .DBYTE, .TEXT, .WORD und *=* +N Pseudo-Operationen besitzen gewöhnlich Markierungen.
3) .END sollte keine Markierung besitzen, da die Bedeutung einer derartigen Markierung unklar ist.

ADRESSEN

ZAHLEN UND ZEICHEN IM ADRESSEN-FELD

Der 6502-Assembler gestattet Eingaben in das Adressenfeld in einer der folgenden Formen:

1) Dezimal (der Standardfall)
   Beispiel: 1247
2) Hexadezimal (muß beginnen mit $)
   Beispiel: $CE00
3) Oktal (muß beginnen mit @)
   Beispiel: @1247
4) Binär (muß beginnen mit %)
   Beispiel: %11100011
5) ASCII (einzelnes Zeichen, dem ein Apostroph vorausgeht)
   Beispiel: 'H
6) Als eine Versetzung vom Befehlszähler (*)
   Beispiel: *+7

ADRESSIER-ARTEN

Die verschiedenen Adressier-Arten des 6502 werden wie folgt unterschieden:

- Absolut oder Null-Seite (direkt) sind die Standard-Adressier-Arten (der Assembler wählt die Null-Seite, wenn die Adresse kleiner als 256 ist, andernfalls die absolute).
- # für unmittelbare Adressierung (geht den Daten voraus).
- ,X oder ,Y für Indizierung (folgt der Versetzungs-Adresse).
- Klammern um die Adresse werden indirekt verwendet, so daß
  (addr,X) eine vor-indizierte (indizierte Adresse, indirekt verwendet) anzeigt
  (addr),Y nach-indizierte (indirekte Adresse ist indiziert) anzeigt
  (addr) nur indirekte mit einem JPM-Befehl anzeigt.

Bei den indizierten Adressier-Arten, wie bei den direkten Adressier-Arten, wählt der Assembler automatisch die Nullseiten-Version, wenn dies zulässig ist und wenn die Adresse kleiner als 256 ist.

ARITHMETISCHE AUSDRÜCKE DES ASSEMBLERS

Der Assembler gestattet auch Ausdrücke im Adressenfeld. Die Ausdrücke bestehen aus Zahlen und Namen, getrennt durch die arithmetischen Operationen +, –, *(Multiplikation) oder / (ganzzahlige Division).
Der Assembler prüft die Ausdrücke von links nach rechts. Es sind weder Klammern zulässig um Operatione zu gruppieren, noch besteht irgendeine Rangfolge von Operationen. Gebrochene Ergebnisse werden gekürzt.

Wir empfehlen, daß Sie Ausdrücke innerhalb des Adressenfeldes wenn immer möglichst vermeiden. Wenn Sie eine Adresse berechnen müssen, kommentieren Sie jeden unklaren Ausdruck und vergewissern Sie sich, daß die Auswertung der Ausdrücke niemals ein Resultat liefert, das zu groß für seine letztliche Anwendung ist.

ANDERE ASSEMBLER-EIGENSCHAFTEN

Der Standard-6502-Assembler besitzt weder eine bedingte Assemblier-Möglichkeit noch eine Makro-Möglichkeit. Einige 6502-Assembler haben eine oder zwei dieser Möglichkeiten, und man sollte die Beschreibung dem entsprechenden Handbuch entnehmen. Wir werden weiterhin keine dieser Möglichkeiten verwenden oder uns hierauf beziehen, obwohl beide sehr bequem in praktischen Anwendungen sein können.

Kapitel 4
EINFACHE PROGRAMME

Der einzige Weg zum Erlernen der Programmierung in Assemblersprache ist das Schreiben von Assemblersprachen-Programmen. Die nächsten sechs Kapitel dieses Buches enthalten Beispiele für einfache Programme von typischen Mikrocomputer-Aufgaben. Sie sollten jedes Beispiel sorgfältig lesen und versuchen, diese Beispiele auf einem Mikrocomputer-System mit dem 6502 auszuführen, um sich zu vergewissern, daß Sie den in dem Kapitel behandelten Stoff auch verstanden haben. Schließlich sollten Sie die Aufgaben am Ende jedes Kapitels durcharbeiten und die sich ergebenden Programme ablaufen lassen. Dieses Kapitel enthält einige sehr einfache Programme.

ALLGEMEINES FORMAT DER BEISPIELE

BEISPIEL-FORMAT

Jedes Programm enthält folgende Teile:

1) Einen Titel, der die Aufgabe allgemein beschreibt.
2) Eine Angabe des Zweckes, wobei die spezielle Aufgabe beschrieben wird, die das Programm ausführt, sowie die verwendeten Speicherplätze.
3) Ein Programmbeispiel, das die Eingabedaten und die Ergebnisse zeigt.
4) Ein Flußdiagramm, wenn die Programmlogik komplex ist.
5) Eine Auflistung des Programms in Form des Quellprogramms oder in Assemblersprache.
6) Die Auflistung des Objektprogramms oder des Programmes in hexadezimaler Maschinensprache.
7) Erklärende Anmerkungen, mit denen die Befehle und die in dem Programm verwendeten Verfahren besprochen werden.

Die Aufgaben am Ende des Kapitels sind ähnlich aufgebaut wie die Beispiele. Die Aufgaben sollten auf einem Mikrocomputer-System mit dem 6502 programmiert werden, wobei die Beispiele als Richtlinie dienen.

Das Quellprogramm in den Beispielen ist wie folgt aufgebaut:

RICHTLINIEN FÜR BEISPIELE

1) Es wird die Standard-Assembler-Schreibweise des 6502 verwendet, wie in Kapitel 3 zusammengefaßt wurde.
2) Die Form, in der Daten und Adressen aufscheinen, sind mehr nach ihrer Deutlichkeit ausgewählt, anstatt nach ihrer Folgerichtigkeit. Wir verwenden hexadezimale Zahlen für Speicheradressen, Befehlscodes und BCD-Daten. Für numerische Konstanten werden Dezimalzahlen verwendet, Binärzahlen für logische Masken und ASCII für Zeichen.
3) Auf häufig verwendete Befehle und Programmier-Techniken wird besonderer Wert gelegt.
4) Die Beispiele illustrieren die Aufgaben, die Mikroprozessoren in der Kommunikation, Instrumentation, Computertechnik, in Bürogeräten und industriellen und militärischen Anwendungen ausführen.

5) Es sind detaillierte Kommentare enthalten.
6) Auf einfache und deutliche Struktur wird besonderer Wert gelegt, die Programme sind jedoch so effizient wie möglich innerhalb dieser Richtlinie. Die Anmerkungen beschreiben häufig effizientere Verfahren.
7) Die Programme verwenden konsequente Speicher-Zuweisungen. Jedes Programm beginnt mit dem Speicherplatz 0000 und endet mit dem Break-Befehl (BRK). Wenn Ihr Mikrocomputer keinen Monitor und keine Unterbrechungen besitzt, können Sie das Programm mit einem Befehl für eine endlose Schleife beenden, wie etwa

```
HERE    JMP    HERE
```

Einige Mikrocomputer mit den 6502 können einen JMP- oder JSR-Befehl mit einer speziellen Bestimmungsadresse zur Steuerung des Monitors erfordern. Andere Mikrocomputer können eine Spezifikation der Monitor-Adresse benötigen, die mit dem BRK-Befehl zu verwenden ist. Wenn Sie beispielsweise den populären KIM-1 verwenden, müssen Sie 1C00 in die Adressen 17FE und 17FF laden. Achten Sie jedoch darauf: die 00 muß in die Adresse 17FE und 1C in die Adresse 17FF geladen werden. Wir werden später erklären, wie der 6502 Adressen speichert und wie er den BRK-Befehl ausführt (siehe Kapitel 12).

Schlagen Sie im Anwender-Handbuch Ihres Mikrocomputers nach, um die erforderlichen Speicherzuweisungen und Abschlußbefehle für Ihr spezielles System zu bestimmen.

RICHTLINIEN ZUR LÖSUNG VON AUFGABEN

Verwenden Sie die folgenden Richtlinien zur Lösung der Aufgaben am Ende jedes Kapitels:

PROGRAMMIER-RICHTLINIEN

1) Kommentieren Sie jedes Programm so, daß andere es verstehen können. Die Kommentare können kurz und ungrammatisch sein.
   Sie sollen den Zweck eines Abschnittes oder Befehls im Programm erklären. Kommentare sollten nicht die Arbeitsweise von Befehlen beschreiben, derartige Beschreibungen sind in den Handbüchern enthalten. Man braucht nicht jede Anweisung zu kommentieren oder erklären, wenn diese offensichtlich sind. Man kann dem Format der Beispiele folgen, braucht jedoch nicht so sehr ins Detail zu gehen.
2) Bevorzugen Sie Klarheit, Einfachheit und einen guten Aufbau der Programme. Wenn auch Programme möglichst effizient sein sollen, so ist es jedoch nicht unbedingt erforderlich, ein einzelnes Wort oder einen Programmspeicherplatz oder einige wenige Mikrosekunden einzusparen.
3) Machen Sie Programme einigermaßen universell, verwechseln Sie nicht Parameter (wie etwa die Nummer eines Elements in einer Anordnung) mit festen Konstanten (wie etwa π oder ASCII C).
4) Nehmen Sie niemals feste Anfangswerte für Parameter an, d.h. verwenden Sie einen Befehl zum Laden eines Anfangswertes in einen Parameter.
5) Verwenden Sie Assembler-Schreibweise, wie sie in den Beispielen gezeigt wird und in Kapitel 3 definiert wurde.

6) Verwenden Sie hexadezimale Schreibweise für Adressen. Verwenden Sie die deutlichste Form für Daten.
7) Wenn Ihr Mikrocomputer es gestattet, beginnen Sie alle Programme im Speicherplatz 0000 und verwenden Sie Speicherplätze beginnend mit 0040 für Daten und zeitweilige Speicherung. Andernfalls richten Sie äquivalente Adressen für Ihren Mikrocomputer ein und verwenden Sie diese konsequent. Die Einzelheiten entnehmen Sie Ihrem Handbuch.
8) Verwenden Sie bedeutungsvolle Namen für Marken und Variable. Zum Beispiel SUM oder CHECK anstatt X, Y, oder Z.
9) Führen Sie jedes Programm auf Ihrem Mikrocomputer aus. Es gibt keinen anderen Weg sich zu vergewissern, daß Ihr Programm korrekt ist. Es wurden bei jeder Aufgabe Daten als Beispiele vorgesehen. Vergewissern Sie sich, daß das Programm auch für spezielle Fälle arbeitet.

Wir wollen nun einige nützliche Informationen zusammenfassen, an die man sich beim Schreiben von Programmen erinnern soll.

VERWENDUNG DES AKKUMULATORS

In nahezu allen Verarbeitungs-Befehlen (z.B. Add, Subtract, AND, OR) verwenden Sie den Inhalt des Akkumulators als *einen* Operanden und speichern die Ergebnisse zurück in den Akkumulator. In den meisten Fällen werden Sie die ursprünglichen Daten in den Akkumulator mit LDA laden. Sie werden in den meisten Fällen das Ergebnis vom Akkumulator in den Speicher mit STA speichern.

VERWENDUNG DER SEITE NULL DES SPEICHERS

Daten, auf die häufig zugegriffen wird, oder Basis-Adressen und Zeiger sollten auf die Nullseite (page zero) des Speichers plaziert werden. Auf diese Daten kann dann mit Null-Seiten- (direkter) vor-indizierter und Null-Seiten-indizierter Adressierung zugegriffen werden.

Beachten Sie speziell, daß sowohl Vor-Indizieren wie Nach-Indizieren, beide annehmen, daß eine Adresse auf der Nullseite gespeichert ist. Die direkte und indizierte Adressier-Art für die Nullseite benötigen beide weniger Zeit und Speicher, als die entsprechenden absoluten Adressier-Arten.

Einige Befehle wie Verschieben, Inkrementieren (Addieren von 1) und Dekrementieren (Subtrahieren von 1) können direkt auf die Daten im Speicher arbeiten. Derartige Befehle gestatten Ihnen, an den Anwender-Registern vorbei zu gehen, benötigen jedoch zusätzliche Ausführungszeit, da die Daten in Wirklichkeit in die CPU geladen und die Ergebnisse in den Speicher zurückglegt werden müssen.

PROGRAMM-BEISPIEL

8-Bit-Datentransfer

Zweck: Bringe den Inhalt des Speicherplatzes 0040 zum Speicherplatz 0041.

Beispiel:

(0040) = 6A
Ergebnis: (0041) = 95

Quellprogramm:

```
        LDA     $40             ;HOLE DATEN
        STA     $41             ;TRANSFERIERE ZU NEUEM
                                ;  SPEICHERPLATZ
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)
0000	A5	LDA $40
0001	40	
0002	85	STA $41
0003	41	
0004	00	BRK

LDA (Load Accumulator) und STA (Store Accumulator) benötigen eine Adresse zur Bestimmung der Quelle oder des Bestimmungsortes der Daten. Da die in diesem Beispiel verwendeten Adressen auf Seite Null liegen (d.h., die acht höchstwertigen Bits sind alles Nullen), kann die (direkte) Null-Seiten-Form der Befehle mit der Adresse im nächsten Wort verwendet werden. Die führenden Nullen können weggelassen werden. Die Adressen sind in Wirklichkeit 0040 und 0041, es kann jedoch die abgekürzte Form verwendet werden, wie es auch im Alltagsleben üblich ist (zB., sagt man "sechzig Pfennig" und nicht "null Mark und sechzig Pfennig").

BRK (Force Break) wird am Ende aller Beispiele verwendet und gibt die Steuerung zum Monitor zurück. Erinnern Sie sich daran, daß Sie gegebenfalls einen entsprechenden Befehl für Ihren Mikrocomputer verwenden müssen.

8-Bit-Addition

Zweck: Addiere den Inhalt der Speicherplätze 0040 und 0041, und plaziere das Ergebnis in den Speicherplatz 0042.

Beispiel:

(0040) = 38
(0041) = 2B
Ergebnis: (0042) = 63

Quellprogramm:

```
        CLC                     ;LÖSCHE ÜBERTRAG ZU BEGINN
        LDA     $40             ;HOLE ERSTEN OPERANDEN
        ADC     $41             ;ADDIERE ZWEITEN OPERANDEN
        STA     $42             ;SPEICHERE ERGEBNIS
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)
0000	18	CLC
0001	A5	LDA $40
0002	40	
0003	65	ADC $41
0004	41	
0005	85	STA $42
0006	42	
0007	00	BRK

Der einzige Additionsbefehl des Mikroprozessors 6502 ist ADC (Addiere mit Übertrag), der in (A) = (A) + (M) + (Übertrag) resultiert, wobei M der adressierte Speicherplatz ist. Wir benötigen daher anfangs den Befehl CLC (Lösche Übertrag), wenn der Wert des Übertrags nicht die Addition beeinflussen soll. Erinnern Sie sich daran, daß der Übertrag in allen Additionen und Subtraktionen enthalten ist.

Die Nullseiten-Form (direkte) aller Befehle wird verwendet, da die Adressen in den ersten 256 Bytes des Speichers liegen.

ADC beeinflußt das Übertragsbit, LDA und STA dagegen nicht. Nur arithmetische und Schiebebefehle beeinflussen den Übertrag, logische und Transfer-Befehle nicht.

LDA und ADC beeinflussen den Inhalt des Speicherplatzes nicht. STA ändert den Inhalt des addressierten Speicherplatzes, hat jedoch keinen Einfluß auf den Inhalt des Akkumulators.

Vergewissern Sie sich, daß das Flag für die Dezimal-Betriebsart (D) gelöscht ist, wenn Sie dieses Programm ausführen. Um absolut sicher zu sein, daß sich das D-Flag im richtigen Zustand befindet, könnten Sie einen CLD-Befehl ($D8_{16}$) beim Beginn des Programmes einsetzen. Wenn Sie einen Mikrocomputer KIM-1 verwenden, sollten Sie den Speicherplatz 00F1 löschen, um sicher zu sein, daß das Flag für die Dezimal-Betriebsart Ihr Programm oder den Monitor nicht stört.

Verschieben um ein Bit nach links

Zweck: Verschiebe den Inhalt des Speicherplatzes 0040 um ein Bit nach links und plaziere das Ergebnis in den Speicherplatz 0041. Lösche die leere Bit-Position.

Beispiel:

(0040) = 6F

Ergebnis: (0041) = DE

Quellprogramm:

```
LDA   $40     ;HOLE DATEN
ASL   A       ;VERSCHIEBE NACH LINKS
STA   $41     ;SPEICHERE ERGEBNIS
BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)
0000	A5	LDA $40
0001	40	
0002	0A	ASL A
0003	85	STA $41
0004	41	
0005	00	BRK

Der Befehl ASL A verschiebt den Akkumulator um ein Bit nach links und löscht das niedrigstwertige Bit. Das höchstwertige Bit gelangt in den Übertrag. Das Ergebnis ist das Zweifache der ursprünglichen Daten (warum?).

Beachten Sie, daß wir auch den Inhalt des Speicherplatzes 0040 um ein Bit mit dem Befehl ASL $40 verschieben und dann das Ergebnis zum Speicherplatz 0041 bringen könnten. Dieses Verfahren würde jedoch den Inhalt des Speicherplatzes 0040 ändern, sowie den Inhalt des Speicherplatzes 0041.

Ausmaskieren der höchswertigen vier Bits

Zweck: Plaziere die niedrigstwertigen vier Bits des Speicherplatzes 0040 in die niedrigstwertigen vier Bits des Speicherplatzes 0041. Lösche die höchstwertigen vier Bits des Speicherplatzes 0041.

Beispiel:

(0040) = 3D

Ergebnis: (0041) = 0D

Quellprogramm:

```
LDAA  $40          ;HOLE DATEN
AND   #%00001111   ;MASKIERE DIE VIER MSBs
STA   $41          ;SPEICHERE ERGEBNIS
BRK
```

Anmerkung: # bedeutet unmittelbare Adressierung und % bedeutet binäre Konstante in der Standard-Assembler-Schreibweise des 6502.

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)
0000	A5	LDA $40
0001	40	
0002	29	AND #%00001111
0003	0F	
0004	85	STA $41
0005	41	
0006	00	BRK

AND # %00001111 UNDiert logisch den Inhalt des Akkumulators mit der Zahl OF_{16}, nicht den Inhalt des Speicherplatzes 000F. Unmittelbare Adressierung (angezeigt durch #) bedeutet, daß die tatsächlichen Daten, nicht die Adresse der Daten, im Befehl enthalten ist.

Die Maske (00001111) wird in binär geschrieben, um ihren Zweck für den Leser deutlicher zu machen. Die binäre Schreibweise für Masken ist klarer als hexadezimale Schreibweise, da logische Operationen eher Bit für Bit ausgeführt werden, anstatt an Ziffern oder Bytes gleichzeitig. Das Ergebnis hängt natürlich nicht von der Programmier-Schreibweise ab. Hexadezimale Schreibweise sollte für Masken verwendet werden, die größer als 8 Bits sind, da dann die binäre Version lang und unübersichtlich wird. Die Kommentare sollten die Maskier-Operationen beschreiben.

Ein logischer AND-Befehl kann zum Löschen von Bits dienen, die nicht verwendet werden. Zum Beispiel könnten die vier niedrigstwertigen Bits von Daten eine Eingabe von einem 10-poligen Schalter sein, oder eine Ausgabe zu einer numerischen Anzeige.

Löschen eines Speicherplatzes

Zweck: Lösche den Speicherplatz 0040.

Quellprogramm:

```
LDA   #
STA   $40          ;LÖSCHE SPEICHERPLATZ 40
BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)
0000	A9	LDA #0
0001	00	
0002	85	STA $40
0003	40	
0004	00	BRK

Null wird nicht anders als andere Zahlen behandelt – der 6502 hat keinen speziellen Löschbefehl. Erinnern Sie sich jedoch daran daß LDA #0 das Nullflag auf eins setzt. Beachten Sie immer diese Logik – das Z (Zero)-Flag wird auf **eins** gesetzt, wenn das letzte Ergebnis **null** war.

STA beeinflußt keinerlei Status-Flags.

Zerlegung eines Wortes

Zweck: Teile den Inhalt des Speicherplatzes 0040 in zwei 4-Bit-Abschnitte auf und speichere sie in die Speicherplätze 0041 und 0042. Plaziere die vier höchstwertigen Bits des Speicherplatzes 0040 in die vier niedrigstwertigen Bit-Positionen des Speicherplatzes 0042. Lösche die vier höchstwertigen Bit-Positionen der Speicherplätze 0041 und 0042.

Beispiele:

(0040) = 3F

Ergebnis: (0041) = 03
(0042) = 0F

Quellprogramm:

```
LDA   $40          ;HOLE DATEN
AND   #%00001111   ;MASKIERE VIER MSBs AUS
STA   $42          ;SPEICHERE LSBs
LDA   $40          ;SPEICHERE DATEN ZURÜCK
LSR   A            ;VERSCHIEBE DATEN 4 MAL LOGISCH
                   ;  NACH RECHTS
LSR   A
LSR   A
LSRA  A
STA   $41          ;SPEICHERE MSBs
BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)
0000	A5	LDA $40
0001	40	
0002	29	AND #%00001111
0003	0F	
0004	85	STA $42
0005	42	
0006	A5	LDA $40
0007	40	
0008	4A	LSR A
0009	4A	LSR A
000A	4A	LSR A
000B	4A	LSR A
000C	85	STA $41
000D	41	
000E	00	BRK

Eine logische Verschiebung um vier Positionen erfordert vier Ausführungen des LSR A-Befehls.

Jeder LSR-Befehl löscht das höchstwertige Bit des Ergebnisses. Daher werden die vier höchstwertigen Bits des Akkumulators alle gelöscht, wenn LSR A viermal ausgeführt wurde.

Sie könnten vielleicht wünschen, das Programm neu zu schreiben, so daß eine Kopie der Daten im Indexregister X aufbewahrt, anstatt die gleichen Daten zweimal zu laden. Welche Version würden Sie vorziehen und weshalb?

Finden der größeren von zwei Zahlen

Zweck: Plaziere den größeren Inhalt der Speicherplätze 0040 und 0041 in den Speicherplatz 0042. Nimm an, daß der Inhalt der Speicherplätze 0040 und 0041 Binärzahlen ohne Vorzeichen sind.

Beispiel:

```
a.          (0040)  =  3F
            (0041)  =  2B
Ergebnis:   (0042)  =  3F

b.          (0040)  =  75
            (0041)  =  A8
Ergebnis:   (0042)  =  A8
```

Quellprogramm:

```
        LDA   $40       ;HOLE DEN ERSTEN OPERANDEN
        CMP   $41       ;IST ZWEITER OPERAND GRÖSSER?
        BCS   STRES
        LDA   $41       ;JA HOLE STATTDESSEN ZWEITEN OPER-
                        ;  ANDEN
STRES   STA   $42       ;SPEICHERE GRÖSSEREN OPERANDEN
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
0000	A5		LDA	$40
0001	40			
0002	C5		CMP	$41
0003	41			
0004	B0		BCS	STRES
0005	02			
0006	A5		LDA	$41
0007	41			
0008	85	STRES	STA	$42
0009	42			
000A	00		BRK	

CMP $41 subtrahiert den Inhalt des Speicherplatzes 0041 vom Inhalt des Akkumulators, speichert das Ergebnis jedoch nicht irgendwohin. Der Befehl wird ausschließlich zum Setzen der Flags für eine folgende bedingte Verzweigung verwendet.

CMP beeinflußt die Flags wie folgt:

1) N nimmt den Wert des höchstwertigen Bits des Ergebnisses der Subtraktion an.
2) Z wird 1, wenn das Ergebnis der Subtraktion 0 ist, andernfalls wird es 0.
3) C wird auf 1 gesetzt, wenn die Subtraktion kein "Borgen" benötigt und auf 0, wenn dies der Fall ist. Beachten Sie, daß C ein invertiertes Borgen ist, nicht das tatsächliche Borgen, wie bei zahlreichen anderen Mikroprozessoren.
4) V wird nicht beeinflußt.

Beachten Sie folgende Fälle:

1) Wenn die Operanden gleich sind, so ist Z = 1. Wenn sie nicht gleich sind, so ist Z = 0.
2) Wenn der Inhalt des Akkumulators größer oder gleich wie der Inhalt der anderen Adresse ist (vorausgesetzt, daß beide Binärzahlen ohne Vorzeichen sind), so ist C = 1, da dann kein Borgen erforderlich wäre. Andernfalls ist C = 0.

Alle Befehle des 6502 für eine bedingte Verzweigung verwenden die relative Adressierung, bei der das zweite Wort des Befehls eine 8-Bit-Zweierkomplement-Zahl ist, die die CPU zur Adresse des nächsten Befehls addiert, um die Ziel-Adresse zu finden. In diesem Beispiel ist die relative Versetzung gleich 0008 (Ziel-Adresse) – 0006 (Adresse, die unmittelbar der Verzweigung folgt) oder 02.

Offensichtlich ist das Berechnen von relativen Versetzungen sehr fehleranfällig, speziell wenn das Ergebnis negativ ist. Wenn Sie jedoch alle Ziel-Befehle markieren, wird der Assembler die Berechnungen für Sie ausführen.

BCS bewirkt eine Verzweigung, wenn der Übertrag 1 ist. Ist der Übertrag 0, so setzt der Prozessor die Ausführung der Befehle in seiner normalen Sequenz fort, so als ob der Verzweigungs-Befehl nicht existieren würde.

STRES ist eine Marke (Label), ein Name, den der Programmierer einer Speicher-Adresse zuweist, so daß man sich an diese leichter erinnern und sie lokalisieren kann. Beachten Sie, daß Markierungen von einem Zwischenraum auf der Zeile gefolgt werden, auf der sie definiert werden. Die Markierung macht den Zielort einer Verzweigung deutlich, speziell wenn relative Adressierung verwendet wird.

Die Verwendung einer Marke ist zu empfehlen, um vor allem die Versetzung zu spezifizieren (d.h. BCS*+4), da die Befehle des 6502 unterschiedliche Länge besitzen. Ein anderer Anwender könnte deshalb leicht einen Fehler bei der Bestimmung einer Versetzung machen.

16-Bit-Addition

Zweck: Addiere die 16-Bit-Zahl in den Speicherplätzen 0040 und 0041 zur 16-Bit-Zahl in den Speicherplätzen 0042 und 0043. Die höchstwertigen acht Bits befinden sich in den Speicherplätzen 0041 und 0043. Speichere das Ergebnis in die Speicherplätze 0044 und 0045, mit den höchstwertigen Bits in 0045.

Beispiel:

```
        (0040) = 2A
        (0041) = 67
        (0042) = F8
        (0043) = 14
Resultat:672A + 14F8 = 7C22
        (0044) = 22
        (0045) = 7C
```

Quellprogramm:

```
    CLC             ;LÖSCHE ÜBERTRAG ZU BEGINN
    LDA   $40       ;ADDIERE DIE NIEDRIGSTWERTIGEN BITS
    ADC   $42
    STA   $44
    LDA   $41       ;ADDIERE DIE HÖCHSTWERTIGEN BITS MIT
                    ;  ÜBERTRAG
    ADC   $43
    STA   $45
    BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)	
0000	18	CLC	
0001	A5	LDA	$40
0002	40		
0003	65	ADC	$42
0004	42		
0005	85	STA	$44
0006	44		
0007	A5	LDA	$41
0008	41		
0009	65	ADC	$43
000A	43		
000B	85	STA	$45
000C	45		
000D	00	BRK	

Sie müssen den Übertrag vor der ersten Addition löschen, da es niemals einen Übertrag in die niedrigstwertigen Bits gibt.

ADC beinhaltet dann automatisch den Übertrag von den niedrigstwertigen Bits in die Addition der höchstwertigen Bits. Der Mikroprozessor kann daher Daten jeder beliebigen Länge addieren. Er addiert Zahlen mit acht Bits gleichzeitig, wobei die Übertrags-Information von einem 8-Bit-Abschnitt zum nächsten übertragen wird. Beachten Sie jedoch, daß jede 8-Bit-Addition die Ausführung von drei Befehlen (LDA, ADC, STA) erfordert, da es nur einen Akkumulator gibt.

Tabelle der Quadrate

Zweck: Berechne das Quadrat des Inhalts des Speicherplatzes 0041 aus einer Tabelle und plaziere ihn in den Speicherplatz 0042. Es wird angenommen, daß der Speicherplatz 0041 eine Zahl zwischen 0 und einschließlich 7 enthält. $(0 \leqslant (0041) \leqslant 7)$

Diese Tabelle besetzt die Speicherplätze 0050 bis 0057.

Speicher-Adresse (Hex)	**Eingabe**	
	(Hex)	**(Dezimal)**
0050	00	0 (0^2)
0051	01	1 (1^2)
0052	04	4 (2^2)
0053	09	9 (3^2)
0054	10	16 (4^2)
0055	19	25 (5^2)
0056	24	36 (6^2)
0057	31	49 (7^2)

Beispiel:

a. (0041) = 03
Ergebnis: (0042) = 09

b. (0041) = 06
Ergebnis: (0042) = 24

Erinnern Sie sich daran, daß das Ergebnis eine Hexadezimal-Zahl ist.

Quellprogramm:

```
        LDX    $41              ;HOLE DATEN
        LDA    $50,X            ;HOLE QUADRAT DER DATEN
        STA    $42              ;SPEICHERE QUADRAT
        BRK
        *=$50                   ;QUADRAT-TABELLE
SQTAB   .BYTE  0,1,4,9,16,25,36,49
```


Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A6		LDX	$41
0001	41			
0002	B5		LDA	$50,X
0003	50			
0004	85		STA	$42
0005	42			
0006	00		BRK	
0050	00	SQTAB	.BYTE	0
0051	01			1
0052	04			4
0053	09			9
0054	10			16
0055	19			25
0056	24			36
0057	31			49

Beachten Sie, daß Sie auch die Tabelle der Quadrate in den Speicher eingeben müssen (die Assembler-Pseudo-Operation .BYTE wird dies durchführen). Die Tabelle der Quadrate sind konstante Daten, nicht Parameter, die sich ändern können. Deshalb können Sie die Tabelle initialisieren, indem Sie die Byte-Pseudo-Operation verwenden, anstatt Befehle auszuführen, um Werte in die Tabelle zu laden. Erinnern Sie sich daran, daß in praktischen Anwendungen, die Tabelle ein Teil des Festwertspeichers sein würde. Die .BYTE-Pseudo-Operation plaziert die spezifizierten Daten in den Speicher in der Reihenfolge, in der sie im Operandenfeld aufscheinen.

Die Pseudo-Operation *= bestimmt einfach, wo der Lader (oder Assembler) den nächsten Abschnitt des Codes plazieren wird, wenn er schließlich in den Speicher des Mikroprozessors für die Ausführung eingegeben wird. Beachten Sie, daß die Pseudo-Operation niemals wirklich in einem zu erzeugenden Objektcode resultiert.

Indizierte Adressierung (oder Indizieren) bedeutet, daß die vom Befehl verwendete tatsächliche Adresse die Summe der Adresse ist, die im Befehl enthalten ist (häufig als effektive Adresse bezeichnet) und dem Inhalt des Indexregisters. Daher ist LDA $50,X (,X oder ,Y zeigt indizierte Adressierung in der Assemblersprache des 6502 an) gleich ist mit LDA $50 + (X), oder LDA $53, wenn (X) = 03. In diesem Programm-Beispiel enthält das Indexregister X die zu quadrierende Zahl, und die Adresse, die in diesem Befehl eingeschlossen ist und stellt die Start-Adresse der Tabelle der Quadrate dar.

Beachten Sie, daß es eine spezielle indizierte Nullseiten-Adressierung unter Verwendung des Indexregisters X gibt.

Indizierung benötigt immer zusätzliche Zeit, da der Mikrocomputer eine Addition zur Berechnung der effektiven Adresse ausführen muß. Daher erfordert LDA $50,X vier Takt-Zyklen, während LDA $50 nur drei braucht. Es würde jedoch offensichtlich wesentlich mehr Zeit erfordern, um auf die Tabellen-Eingabe zuzugreifen, wenn der Mikroprozessor kein Indizieren besäße und die Adressen-Berechnung mit einer Serie von Befehlen auszuführen wäre.

Erinnern Sie sich daran, daß die Indexregister nur 8 Bits lang sind, so daß die maximale Versetzung von der Basis-Adresse 255 (FF_{16}) beträgt. Beachten Sie auch, daß die Versetzung eine Zahl ohne Vorzeichen ist (anders als die Versetzung bei der relativen Adressierung), so daß sie niemals negativ sein kann.

Es gibt einige wenige spezielle Befehle, die auf eines der Indexregister arbeiten, anstatt auf einen Akkumulator. Diese sind:

CPX, CPY – Vergleiche Speicher und Indexregister
DEX, DEY – Dekrementiere Indexregister (um 1)
INX, INY – Inkrementiere Indexregister (um 1)
LDX, LDY – Lade Indexregister vom Speicher
STX, STY – Speichere Indexregister in Speicher
TAX, TAY – Transferiere Akkumulator zu Indexregister
TXA, TYA – Transferiere Indexregister zu Akkumulator

Erinnern Sie sich daran, daß es nur einige wenige Adressier-Arten mit CPX, CPY, LDX, LDY, STX und STY gibt. Sehen Sie sich Tabelle 3-4 für weitere Details an.

ARITHMETIK MIT TABELLEN

Arithmetische Aufgaben, die ein Mikroprozessor nicht direkt mit einigen wenigen Befehlen ausführen kann, werden häufig am besten mit "Nachschlage"-Tabellen ausgeführt. Nachschlage-Tabellen (lookup tables) enthalten einfach alle möglichen Antworten für eine Aufgabe. Sie sind so organisiert, daß die Lösung einer speziellen Aufgabe leicht gefunden werden kann. Die arithmetische Aufgabe wird nun zu einer Zugriff-Aufgabe: Wie kann man die richtige Antwort aus der Tabelle erhalten? Wir müssen zwei Dinge wissen: Die Lage oder Position der Antwort in der Tabelle (genannt der Index) und die Basis- oder Start-Adresse der Tabelle. Die Adresse der Antwort ist dann die Basis-Adresse plus dem Index.

Die Basis-Adresse ist natürlich eine feste Zahl für eine bestimmte Tabelle. Wie kann man nun den Index bestimmen? In einfachen Fällen, bei denen nur kurze Daten verwendet werden, kann man die Tabelle so organisieren, daß die Daten den Index darstellen. In der Tabelle der Quadrate enthält die "nullte" Eingabe in die Tabelle Null zum Quadrat, die erste Eingabe Eins zum Quadrat usw. In komplexeren Fällen, bei denen eine größere Vielfalt von Eingangswerten vorhanden ist, oder wo verschiedene Arten von Daten vorkommen (z.B. Wurzeln einer quadratischen Gleichung oder Anzahl der Permutationen) muß man kompliziertere Verfahren zur Bestimmung der Indizes verwenden.

Der grundlegende Vorteil bei der Verwendung einer Tabelle liegt in der Einsparung der Zeit, wobei jedoch mehr Speicherplätze benötigt werden. Tabellen sind schneller, da keine Berechnungen erforderlich sind und einfacher, da keine mathematischen Methoden abgeleitet und getestet werden müssen. Tabellen können jedoch viel Platz im Speicher belegen, wenn der Bereich der Eingangsdaten groß ist. Man kann häufig die Größe einer Tabelle verringern, indem man die Genauigkeit der Ergebnisse reduziert, durch Skalieren der Eingangsdaten, oder einer geschickten Organisation der Tabelle. Tabellen werden häufig zur Berechnung transdezenter und trigonometrischer Funktionen verwendet, zur Linerarisierung von Eingangswerten, zur Umwandlung von Codes und zur Ausführung anderer mathematischer Aufgaben.

Einerkomplement

Zweck: Komplementiere logisch den Inhalt des Speicherplatzes 0040 und plaziere das Ergebnis in den Speicherplatz 0041.

Beispiel:

(0040) = A
Ergebnis: (0041) = 95

Quellprogramm:

```
LDA  $40         ;HOLE DATEN
EOR  #%11111111  ;KOMPLEMENTIERE DATEN LOGISCH
STA  $41         ;SPEICHERE ERGEBNIS
BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)
0000	A5	LDA $40
0001	40	
0002	49	EOR #%11111111
0003	FF	
0004	85	STA $41
0005	41	
0006	00	BRK

Dem Mikroprozessor 6502 fehlen einige einfache Befehle, wie Löschen oder Komplementieren, die in den meisten übrigen Befehlssätzen enthalten sind. Die erforderlichen Operationen können jedoch mit den bestehenden Befehlen leicht ausgeführt werden, wenn man einige einfache Tricks anwendet.

Exklusiv-ODERieren eines Bits mit "1" komplementiert das Bit, da

$$1 \veebar 0 = 1$$

und

$$1 \veebar 1 = 0$$

Auf diese Weise verwandelt die Exklusiv-ODER-Funktion jedes 0-Bit in eine 1 und ein 1-Bit in eine 0, so daß dies einem logischen Komplementieren oder Invertieren entspricht. Beachten Sie jedoch, daß der Befehl EOR #%11111111 zwei Speicherbytes belegt, eines für den Operationscode und eines für die Maske. Ein spezieller Komplementier-Befehl würde nur ein Byte benötigen.

Ein Problem mit dieser Lösung besteht darin, daß der Zweck des Befehls nicht unmittelbar zu ersehen ist. Ein Leser würde wahrscheinlich überlegen, was eine Exklusiv-ODER-Funktion mit einem Wort aus lauter Einsen wirklich ausführt. Eine entsprechende Dokumentation ist hier sehr wesentlich, und die Verwendung von Makros kann ebenfalls zum Klären der Situation beitragen.

AUFGABEN

1) 16-Bit-Daten-Transfer

Zweck: Bringe den Inhalt des Speicherplatzes 0040 zum Speicherplatz 0042 und den Inhalt des Speicherplatzes 0041 zum Speicherplatz 0043.

Beispiel:

	(0040) = 3E
	(0041) = 87
Ergebnis:	(0042) = 3E
	(0043) = B7

2) 8-Bit-Subtraktion

Zweck: Subtrahiere den Inhalt des Speicherplatzes 0041 vom Inhalt des Speicherplatzes 0040. Plaziere das Ergebnis in den Speicherplatz 0042.

Beispiel:

	(0040) = 77
	(0041) = 39
Ergebnis:	(0042) = 3E

3) Verschiebe um zwei Bits nach links

Zweck: Verschiebe den Inhalt des Speicherplatzes 0040 nach links um zwei Bits und plaziere das Ergebnis in den Speicherplatz 0041. Lösche die beiden niedrigstwertigen Bit-Positionen

Beispiel:

	(0040) = 5D
Ergebnis:	(0041) = 74

4) Maskiere die niedrigstwertigen vier Bits aus

Zweck: Plaziere die vier höchstwertigen Bits des Inhalts des Speicherplatzes 0040 in den Speicherplatz 0041. Lösche die vier niedrigstwertigen Bits des Speicherplatzes 0041.

Beispiel:

	(0040) = C4
Ergebnis:	(0041) = C0

5) Setze einen Speicherplatz auf Einsen

Zweck: Der Speicherplatz 0040 wird auf lauter Einsen (FF_{16}) gesetzt.

6) Wort-Zusammensetzung

Zweck: Kombiniere die vier niedrigstwertigen Bits der Speicherplätze 0040 und 0041 zu einem Wort und speichere dieses in den Speicherplatz 0042. Plaziere die vier niedrigstwertigen Bits des Speicherplatzes 0040 in die vier höchstwertigen Bit-Positionen des Speicherplatzes 0042. Plaziere die vier niedrigstwertigen Bits des Speicherplatzes 0041 in die vier niedrigstwertigen Bit-Positionen des Speicherplatzes 0042.

Beispiel:

	(0040) = 6A
	(0041) = B3
Ergebnis:	(0042) = A3

7) Auffinden der kleineren von zwei Zahlen

Zweck: Plaziere den kleineren Inhalt der Speicherplätze 0040 und 0041 in den Speicherplatz 0042. Nimm an, daß 0040 und 0041 Binärzahlen ohne Vorzeichen enthalten.

Beispiel:

a.	(0040) = 3F
	(0041) = 2B
Ergebnis:	(0042) = 2B
b.	(0040) = 75
	(0041) = A8
Ergebnis:	(0042) = 75

8) 24-Bit-Addition

Zweck: Addiere die 24-Bit-Zahl in den Speicherplätzen 0040, 0041 und 0042 zu der 24-Bit-Zahl in den Speicherplätzen 0043, 0044 und 0045. Die höchstwertigen 8 Bits befinden sich in den Speicherplätzen 0042 und 0045, die niedrigstwertigen 8 Bits in den Speicherplätzen 0040 und 0043. Speichere das Ergebnis in die Speicherplätze 0046, 0047 und 0048 mit den höchstwertigen Bits in 0048 und den niedrigstwertigen Bits in 0046.

Beispiel:

	(0040) = 2A
	(0041) = 67
	(0042) = 35
	(0043) = F8
	(0044) = A4
	(0045) = 51
Ergebnis:	(0046) = 22
	(0047) = 0C
	(0048) = 87

das heißt:
```
  35672A
 +51A458
  870C22
```

9) Summe der Quadrate

Zweck: Berechne die Quadrate des Inhaltes der Speicherplätze 0040 und 0041 und addiere sie. Plaziere das Ergebnis in den Speicherplatz 0042. Nimm an, daß die beiden Speicherplätze 0040 und 0041 Zahlen zwischen 0 und einschließlich 7 enthalten ($0 \leqslant (0040) \leqslant 7$ und $0 \leqslant (0041) \leqslant 7$). Verwende die Tabelle der Quadrate aus dem Beispiel "Tabelle der Quadrate".

Beispiel:

	(0040)	= 03
	(0041)	= 06
Ergebnis:	(0042)	= 2D

das heißt $3^2 + 6^2 = 19 + 36$ (dezimal)
$= 45 = 2D_{16}$

10) Zweierkomplement

Zweck: Plaziere das Zweier-Komplement des Inhalts des Speicherplatzes 0040 in den Speicherplatz 0041. Das Zweier-Komplement ist das Einer-Komplement plus eins.

Beispiel:

	(0040)	= 3E
Ergebnis:	(0041)	= C2

Die Summe der ursprünglichen Zahl und ihres Zweier-Komplements ist null. Daher ist das Zweier-Komplement von X gleich 0-X. Welche Lösung (Berechnung des Einer-Komplements und Addieren von eins, oder Subtrahieren von null) ergibt ein kürzeres und schnelleres Programm?

Kapitel 5
EINFACHE PROGRAMM-SCHLEIFEN

Die Programmschleife ist die grundlegende Struktur, die die CPU zur Wiederholung einer Folge von Befehlen zwingt. Schleifen besitzen vier Abschnitte:

1) **Der Start- oder Initialisierungs-Abschnitt,** der den Anfangswert der Zähler, Indizes, Zeiger und andere Variablen einrichtet.
2) **Der Verarbeitungs-Abschnitt,** in dem die tatsächliche Daten-Manipulation auftritt. Dieser Abschnitt verrichtet die eigentliche Arbeit.
3) **Der Schleifen-Steuerabschnitt,** der die Zähler und Indizes für die nächste Wiederholung auf den entsprechenden Stand bringt.
4) **Der Abschluß-Abschnitt,** der die Ergebnisse analysiert und speichert.

Es ist zu beachten, daß der Computer die Abschnitte 1 und 4 nur einmal durchläuft, während dies bei den Abschnitten 2 und 3 mehrmals der Fall sein kann. Daher hängt die Ausführungszeit der Schleife im wesentlichen von der Ausführungszeit der Abschnitte 2 und 3 ab. Man wird daher trachten, daß die Abschnitte 2 und 3 so rasch wie möglich ausgeführt werden. Die Ausführungszeit der Abschnitte 1 und 4 sind dagegen zu vernachlässigen. Eine typische Programmschleife kann in einem Flußdiagramm gemäß Bild 5-1 dargestellt werden, oder es kann die Lage der Verarbeitungs- und Schleifen-Steuer-Abschnitte umgekehrt werden, wie in Bild 5-2 gezeigt wird. Der Verarbeitungs-Abschnitt in Bild 5-1 muß immer wenigstens einmal ausgeführt werden, während der Verarbeitungs-Abschnitt in Bild 5-2 auch überhaupt nicht ausgeführt werden muß. Bild 5-1 scheint natürlicher zu sein, jedoch Bild 5-2 ist häufig effizienter und vermeidet Probleme, falls keine Daten vorliegen (ein Schreckgespenst für Computer, und ein häufiger Grund für verrückte Situationen, wenn ein Computer eine Rechnung von 0.00 DM eintreiben soll).

Die Struktur der Schleife kann zur Verarbeitung ganzer Datenblöcke verwendet werden. Um dies auszuführen, muß das Programm das Indexregister nach jeder Wiederholung inkrementieren, so daß die effektive Adresse eines indizierten Befehls das nächste Element im Datenblock ist. Die nächste Wiederholung wird dann die gleichen Operationen an den Daten in der nächsten Speicherstelle ausführen. Der Computer kann Blöcke beliebiger Länge (bis zu 256, da die Indexregister 8-Bits lang sind) mit dem gleichen Befehlssatz hand haben. Die indizierte Adressierung ist der Schlüssel zur Verarbeitung von Datenblöcken mit dem 6502, da er uns gestattet, die momentane (oder effektive) Speicher-Adresse durch Änderung des Inhalts von Index-Registern zu variieren. Man beachte, daß bei den direkten und unmittelbaren Adressier-Verfahren die verwendete Adresse vollständig durch den Befehl bestimmt wird und daher fest liegt, wenn der Programmspeicher ein Festwertspeicher ist.

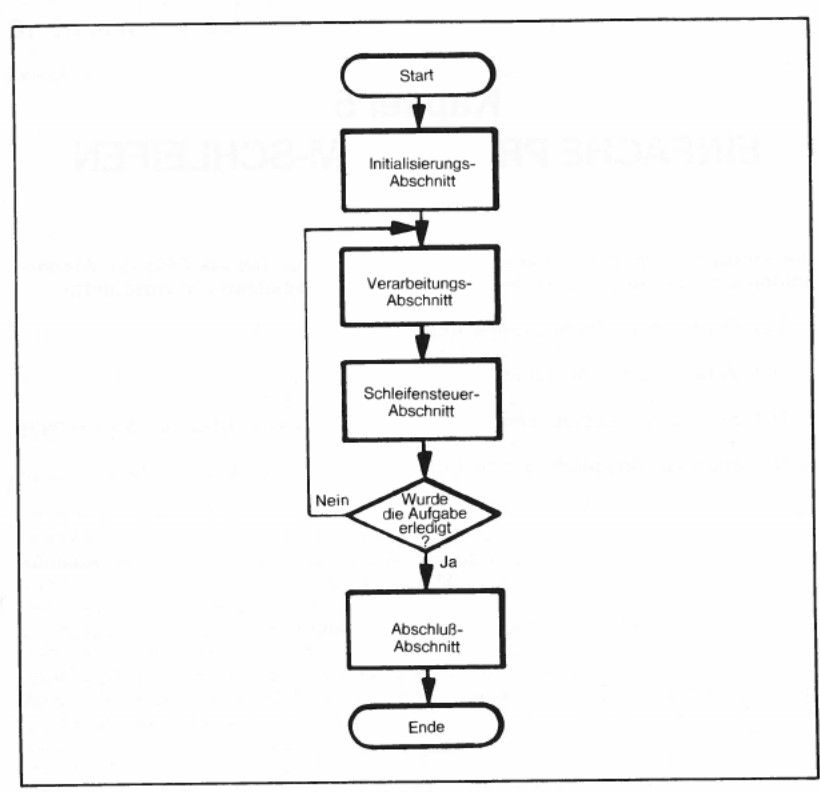


Bild 5-1. Flußdiagramm einer Programm-Schleife

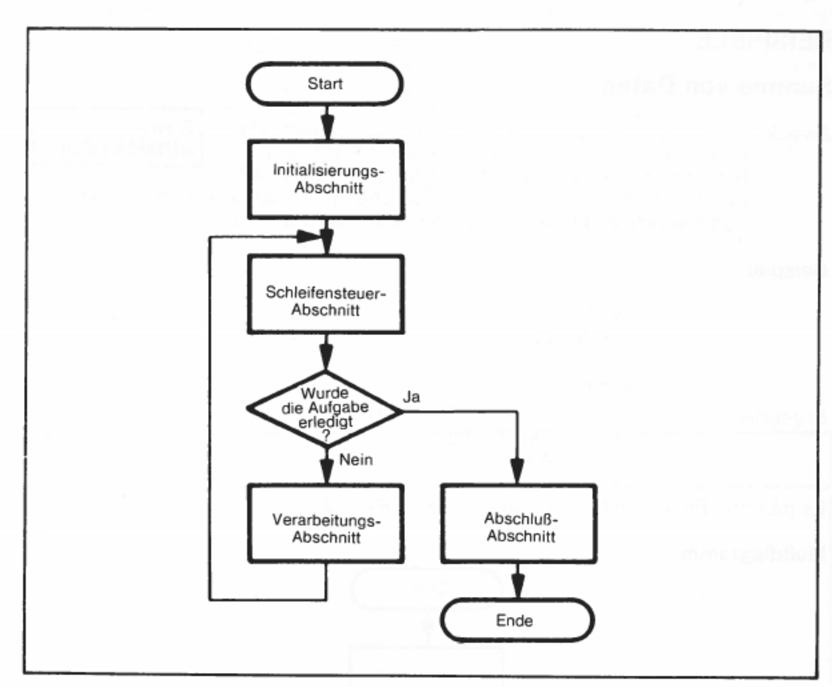


Bild 5-2. Eine Programm-Schleife, die null Wiederholungen gestattet

BEISPIELE

Summe von Daten

8-BIT-SUMMATION

Zweck: Berechne die Summe einer Serie von Zahlen. Die Länge der Serie ist im Spei cherplatz 0041 enthalten und die Serie beginnt im Speicherplatz 0042. Speichere die Summe in den Speicher platz 0040. Nimm an, daß die Summe eine 8-Bit-Zahl ist, so daß man Überträge ignorieren kann.

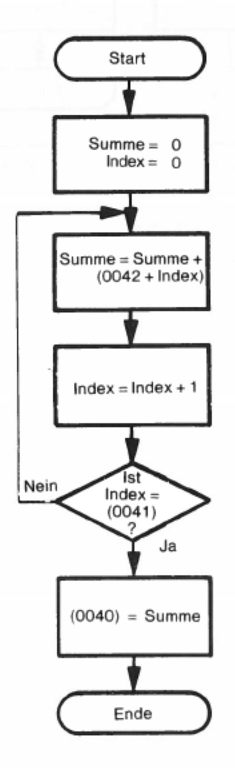
Beispiel:

(0041) = 03
(0042) = 28
(0043) = 55
(0044) = 26

Ergebnis: (0040) = (0042) + (0043) + (0044)
= 28+55 + 26
= A3

Es gibt drei Eingaben für die Summe, da (0041) = 03.

Flußdiagramm:



Anmerkung: (0042+Index) ist der Inhalt des Speicherplatzes dessen Adresse die Summe von 0042 und des Index ist. Erinnern Sie sich daran, daß beim 6502 0042 eine 16-Bit-Adresse ist, der Index ist eine 8-Bit-Versetzung, und (0042+Index) ist ein Datenbyte mit 8 Bits.

```
        LDA   #0      ;SUMME = NULL
        TAX           ;INDEX = NULL
SUMD    CLC           ;ÜBERTRAG NICHT EINSCHLIESSEN
        ADC   $42,X   ;SUMME = SUMME + DATEN
        INX           ;INKREMENTIERE INDEX
        CPX   $41     ;WURDEN ALLE ELEMENTE SUMMIERT ?
        BNE   SUMD    ;NEIN, SETZE SUMMIEREN FORT
        STA   $40     ;JA, SPEICHERE SUMME
        BRK
```

Objektprogramm

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A9		LDA	#0
0001	00			
0002	AA		TAX	
0003	18	SUMD	CLC	
0004	75		ADC	$42,X
0005	42			
0006	E8		INX	
0007	E4		CPX	$41
0008	41			
0009	D0		BNE	SUMD
000A	F8			
000B	85		STA	$40
000C	40			
000D	00		BRK	

Der Initialisierungsabschnittdes Programmes besteht aus den beiden ersten Befehlen, die die Summe und den Index auf ihre Anfangswerte setzen. Beachten Sie, daß TAX den Inhalt des Akkumulators zum Indexregister X transferiert, jedoch den Akkumulator auf seinem ursprünglichen Wert beläßt. Die Basisadresse der Anordnung und der Speicherplatz des Zählers sind innerhalb des Programmes festgelegt und müssen nicht initialisiert werden.

Der Verarbeitungs-Abschnitt besteht aus dem einzelnen Befehl ADC $42,X, der den Inhalt der effektiven Adresse (Basisadresse plus Indexregister X) zum Inhalt des Akkumulators addiert. Dieser Befehl führt die tatsächliche Arbeit des Programmes aus. Der CLC-Befehl löscht einfach das Übertrags-Flag, so daß es die Summation nicht beeinflußt. Beachten Sie, daß jede Wiederholung der Schleife eine Addition zum Inhalt einer neuen effektiven Adresse bedeutet, auch wenn sich die Befehle ändern.

Der Schleifen-Steuerabschnitt des Programmes besteht aus dem Befehl INX. Dieser Befehl bringt das Indexregister auf den neuesten Stand (um 1), so daß die nächste Wiederholung die nächste Zahl zur Summe addiert. Beachten Sie, daß (0041) – X Ihnen sagt, wieviele Wiederholungen noch auszuführen sind.

Der Befehl BNE bewirkt eine Verzweigung, wenn das Nullflag gleich 0 ist. CPX setzt das Nullflag auf 1, wenn das Indexregister X und der Inhalt des Speicherplatzes 0040 gleich sind und 0, wenn dies nicht der Fall ist. Die Versetzung ist eine Zweierkomplement-Zahl und die Zählung beginnt von dem Speicherplatz, der unmittelbar dem BNE-Befehl folgt. In diesem Fall geht der erforderliche Sprung vom Speicherplatz 000B zum Speicherplatz 0003. Daher ist die Versetzung:

```
 0003 =  03
-0008 = +F5
         F8
```

Wenn das Nullflag 1 ist, führt die CPU den nächsten Befehl in der Sequenz (STA $40) aus. Da CPX $41 der letzte Befehl vor BNE war, der das Nullflag beeinflußte, bewirkt BNE SUMD eine Verzweigung zu SUMD, wenn CPX $41 nicht ein Null-Ergebnis erzeugte,
d.h.

$$(PC) = \begin{cases} \text{SUMD wenn } (X) - (0041) \neq 0 \\ (PC) + 2 \text{ wenn } (X) - (0041) = 0 \end{cases}$$

Die 2 wird durch den Zweiwort-Befehl BNE bewirkt. Ein einzelner Befehl, der das Dekrementieren und den Sprung kombinieren würde, wäre eine sehr nützliche Ergänzung zum Befehlssatz des 6502.

Die Reihenfolge, in der die Befehle ausgeführt werden, ist häufig sehr wichtig. INX muß nach ADC $42,X kommen, oder die erste zur Summe zu addierende Zahl wäre der Inhalt des Speicherplatzes 0043 anstatt des Inhalts des Speicherplatzes 0042. CPX $41,X muß rechtzeitig von BNE SUMD kommen, da andernfalls der Nullstatus, der durch CPX erzeugt wird, durch einen anderen Befehl geändert werden könnte.

CPX und CPY sind das gleiche wie CMP, mit Ausnahme, daß der Inhalt des Speicherplatzes von einem Indexregister subtrahiert wird, anstatt vom Akkumulator. Beachten Sie jedoch, daß CPX und CPY begrenzte Adressiermöglichkeiten haben (siehe Tabelle 3-4).

Die meisten Computerschleifen zählen abwärts anstatt aufwärts, so daß das Nullflag als eine Austrittsbedingung dienen kann, wodurch sich ein Vergleichsbefehl erübrigt. Dieses Verfahren ist etwas ungewohnt, obwohl es gelegentlich bei einem Countdown und in einigen anderen Situationen verwendet wird. Erinnern Sie sich daran, daß das Nullflag auf 1 gesetzt wird, wenn das Ergebnis eines Befehl 0 ist, und auf 0, wenn das Ergebnis nicht 0 war.

Wir könnten die Schleife leicht so ändern, daß sie rückwärts durch die Anordnung arbeitet (siehe das nächste Flußdiagramm). Das folgende Programm ist die entsprechend revidierte Version:

Quellprogramm:

```
        LDA   #0       ;SUMME = 0
        LDX   $41      ;INDEX = MAXIMALE ZÄHLUNG
SUMD    CLC            ;SCHLIESSE ÜBERTRAG NICHT EIN
        ACD   $41,X    ;SUMME = SUMME + DATEN
        DEX            ;DEKREMENTIERE INDEX
        BNE   SUMD     ;VERZWEIGE ZURÜCK WENN ALLE ELE
                       ;  MENTE NOCH NICHT SUMMIERT SIND
        STA   $40      ;SPEICHERE SUMME
        BRK
```

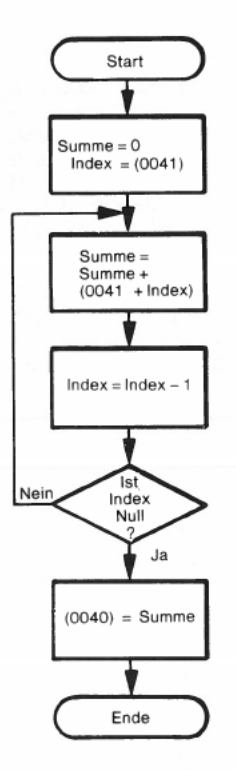
Beachten Sie, daß der Additionsbefehl nun ADC $41,X lautet, anstatt ADC $42,X. Die Zahl im Indexregister ist um 1 größer als vorher. Natürlich ist das Netto-Ergebnis der Subtraktion von 1 von der Basis-Adresse und dem Addieren von eins zum Indexregister gleich null. Das neu organisierte Objektprogramm lautet:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A9		LDA	#0
0001	00			
0002	A6		LDX	$41
0003	41			
0004	18	SUMD	CLC	
0005	75		ADC	$41,X
0006	41			
0007	CA		DEX	
0008	D0		BNE	SUMD
0009	FA			
000A	85		STA	$40
000B	40			
000C	00		BRK	

Bei den meisten Anwendungen sind die geringfügigen Unterschiede in Zeit und Speicher zwischen einer Ausführung einer Schleife und einer anderen nicht von großer Bedeutung. Sie sollten daher die Lösung wählen, die am deutlichsten und leichtesten anzuwenden ist. Wir werden Programm-Entwicklung und Effizienz später in den Kapiteln 13 und 15 besprechen.

Sie könnten vielleicht wünschen, die Hexadezimalwerte für die relativen Versetzungen in den beiden letzten Programmen zu verifizieren. Der Abschlußtest jeder Berechnung besteht im Überprüfen des ordnungsgemäßen Ablaufes eines Programmes. Wenn, aus welchem Grund auch immer, Sie hexadezimale Berechnungen häufig ausführen müssen, so nehmen wir an, daß Sie über einen Rechner (wie dem programmierbaren Rechner von Texas Instruments) oder über eines der zahlreichen Handbücher für diesen Zweck verfügen.

Flußdiagramm: (des reorganisierten Summations-Programmes)



16-Bit-Summe von Daten

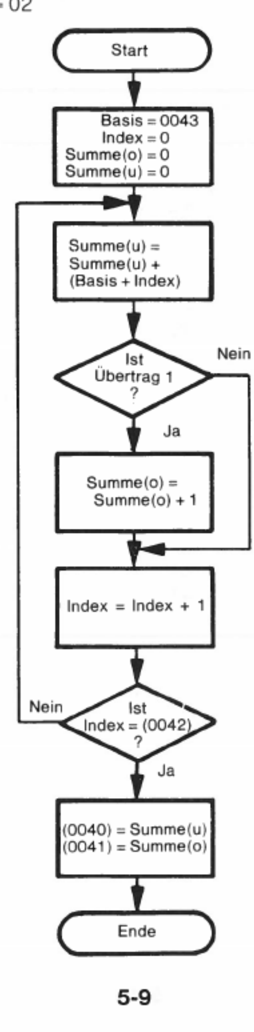
Zweck: Berechne die Summe einer Serie von Zahlen. Die Länge der Serie befindet sich im Speicherplatz 0042 und die Serie selbst beginnt im Speicherplatz 0043. Speichere die Summe in den Speicherplätzen 0040 und 0041 (die acht niedrigstwertigen Bits in 0040).

Beispiel:

(0042) = 03
(0043) = C8
(0044) = FA
(0045) = 96

Ergebnis: C8 + FA + 96 = 0258_{16}
(0040) = 58
(0041) = 02

Flußdiagramm:



Objektprogramm:

```
        LDA   #0        ;SUMME = NULL
        TAX             ;INDEX = NULL
        TAY             ;MSBS DER SUMME = NULL
SUMD    CLC             ;SCHLIESSE ÜBERTRAG NICHT EIN
        ADC   $43,X     ;SUMME = SUMME + DATEN
        BCC   COUNT
        INY             ;ADDIERE ÜBERTRAG ZU MSBS DER SUM-
                        ;  ME
COUNT   INX
        CPX   $42
        BNE   SUMD      ;SETZE FORT BIS ALLE ELEMENTE AD-
                        ;  DIERT SIND
        STA   $40       ;SPEICHERE LSBS DER SUMME
        STY   $41       ;SPEICHERE MSBS DER SUMME
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A9		LDA	#0
0001	00			
0002	AA		TAX	
0003	A8		TAY	
0004	18	SUMD	CLC	
0005	75		ADC	$43,X
0006	43			
0007	90		BCC	COUNT
0008	01			
0009	C8		INY	
000A	E8	COUNT	INX	
000B	E4		CPX	$42
000C	42			
000D	D0		BNE	SUMD
000E	F5			
000F	85		STA	$40
0010	40			
0011	84		STY	$41
0012	41			
0013	00		BRK	

Der Aufbau dieses Programms ist der gleiche wie der Aufbau des vorhergehenden. Die höchstwertigen Bits der Summe müssen nun initialisiert und gespeichert werden. Der Verarbeitungs-Abschnitt besteht aus vier Befehlen (CLC, ADC $43,X, BCC COUNT; und INY), einschließlich eines bedingten Sprunges.

BCC COUNT bewirkt einen Sprung zum Speicherplatz COUNT, wenn Übetrag = 0. Daher, wenn es keinen Übertrag aus der 8-Bit-Addition gibt, springt das Programm über die Anweisung, die die höchstwertigen Bits der Summe inkrementieren. Die relative Versetzung ist

```
 000A
-0009
   01
```

Die relative Versetzung für BNE SUMD ist

```
 0004    0004
      =
-000F   +FFF1
           F5
```

INY addiert 1 zum Inhalt des Indexregisters X, das hier als zeitweiliges Register zum Aufbewahren der Überträge aus der Addition dient. Wir könnten auch einen Speicherplatz zum Aufbewahren der Überträge verwenden, da der INC-Befehl zum direkten Inkrementieren des Inhaltes eines Speicherplatzes eingesetzt werden kann.

Sie möchten vielleicht versuchen, dieses Programm neu so anzuordnen, daß es den Index abwärts auf null dekrementiert, anstatt ihn zu inkrementieren. Welche Version ist kürzer und schneller?

LANGE BEDINGTE VERZWEIGUNGEN

Relative Verzweigungen sind auf kurze Entfernungen beschränkt ($7F_{16}$ oder +127 vorwärts, 80_{16} oder −128 rückwärts vom Ende des Verzweigungsbefehls). Diese Begrenzung ist selten von Bedeutung, da die meisten Programm-Verzweigungen kurz sind.

Wenn Sie jedoch eine bedingte Verzweigung mit einem größeren Bereich benötigen, können Sie immer die Bedingungslogik invertieren und die Verzweigung über einen JMP-Befehl ausführen. Um zum Beispiel zum Speicherplatz FAR zu verzweigen, wenn Übertrag = 0, verwenden Sie die Sequenz

```
        BCS   NEXT
        JMP   FAR
NEXT
```

NEXT ist die Adresse, die unmittelbar dem letzten Byte des JMP-Befehls folgt. JMP gestattet nur absolute (direkte) und indirekte Adressierung.

Anzahl der negativen Elemente

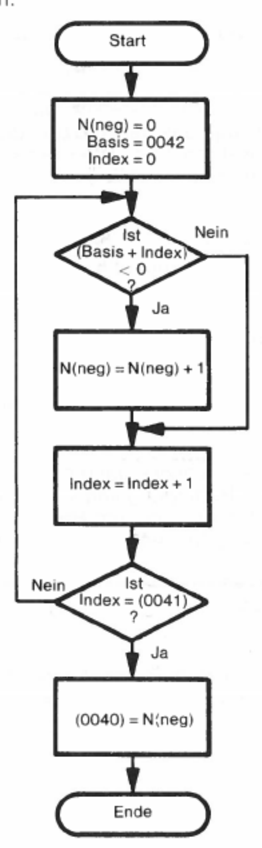
Zweck: Bestimme die Anzahl der negativen Elemente (höchstwertiges Bit 1) in einem Block. Die Länge des Blocks befindet sich im Speicherplatz 0041 und der Block selbst beginnt im Speicherplatz 0042. Plaziere die Anzahl der negativen Elemente in den Speicherplatz 0040.

Beispiel:

(0041) = 06
(0042) = 68
(0043) = F2
(0044) = 87
(0045) = 30
(0046) = 59
(0047) = 2A

Ergebnis: (0040) = 02, da 0043 und 0044 Zahlen mit einem MSB von 1 enthalten.

Flußdiagramm:



Quellprogramm:

```
         LDX   #0        ;INDEX = NULL
         LDY   #0        ;ANZAHL DER NEGATIVEN = NULL
SRNEG    LDA   $42,X     ;IST DAS NÄCHSTE ELEMENT NEGATIV?
         BPL   CHCNT
         INY             ;JA, ADDIERE 1 ZUR NEGATIVZÄHLUNG
         CHCNT INX
         CPX   $41
         BNE   SRNEG     ;SETZE FORT, BIS ALLE ELEMENTE GE-
                         ;  PRÜFT SIND
         STY   $40       ;BEWAHRE NEGATIVZÄHLUNG AUF
         BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A2		LDX	#0
0001	00			
0002	A0		LDY	#0
0003	00			
0004	B5	SRNEG	LDA	$42,X
0005	42			
0006	10		BPL	CHCNT
0007	01			
0008	C8		INY	
0009	E8	CHCNT	INX	
000A	E4		CPX	$41
000B	41			
000C	D0		BNE	SRNEG
000D	F6			
000E	84		STY	$40
000F	40			
0010	00		BRK	

LDA beeinflußt das Negativ-(N)- und Null-(Z)-Statusflag. Daher können wir unmittelbar prüfen um festzustellen, ob eine Zahl, die geladen wurde, negativ oder null ist.
BPL (Branch-on-Plus) bewirkt eine Verzweigung über die spezifizierte Anzahl von Speicherplätzen, wenn das Vorzeichen (oder Negativ)-Bit gleich null ist. Ein Vorzeichen-Bit von 0 kann eine positive Zahl anzeigen oder einfach den Wert der höchstwertigen Bitposition angeben. Die Interpretation hängt davon ab, was die Zahl bedeutet.

Die Versetzung für BPL wird vom ersten Speicherplatz berechnet, der dem Zweiwort-Befehl folgt. Hier reicht die Versetzung einfach von 0008 bis 0009, oder einen Speicherplatz (d.h. der INY-Befehl wird übersprungen, wenn das Negativ-Bit null ist). Das Negativ-Bit wird null sein, wenn das höchstwertige Bit der Daten, die vom Speicher durch den Befehl LDA $42,X geladen werden, null ist.

Erinnern wir uns daran, daß Zahlen mit negativen Vorzeichen immer ein höchstwertiges Bit (Bit 7) von 1 besitzen. Alle negativen Zahlen sind tatsächlich größer (im Sinne einer Betrachtung ohne Vorzeichen) als positive Zahlen.

Finden des Maximums

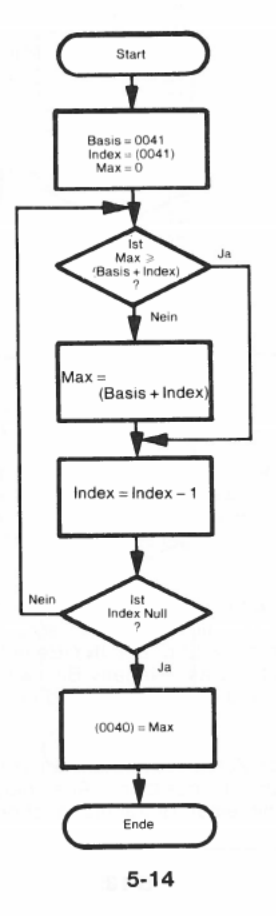
Zweck: Finde das größte Element in einem Datenblock. Die Länge des Blocks befindet sich im Speicherplatz 0041 und der Block selbst beginnt im Speicherplatz 0042. Speichere das Maximum im Speicherplatz 0040. Nimm an, daß die Zahlen im Block 8-Bit-Binärzahlen ohne Vorzeichen sind.

Beispiel:

(0041) = 05
(0042) = 67
(0043) = 79
(0044) = 15
(0045) = E3
(0046) = 72

Ergebnis: (0040) = E3, da dies die größte der fünf Zahlen ohne Vorzeichen ist.

Flußdiagramm:



Quellprogramm:

```
        LDX   $41     ;HOLE ZÄHLUNG
        LDA   #0      ;MAXIMUM = NULL (KLEINSTMÖGLICHER
                      ;   WERT)
MAXM    CMP   $41,X   ;IST NÄCHSTES ELEMENT ÜBER MAXI-
                      ;   MUM?
        BCS   NOCHG   ;JA, BEWAHRE MAXIMUM AUF
        LDA   $41,X   ;NEIN, ERSETZE MAXIMUM DURCH ELE-
                      ;   MENT
NOCHG   DEX
        BNE   MAXM    ;SETZE FORT BIS ALLE ELEMENTE GE-
                      ;   PRÜFT SIND
        STA   $40     ;BEWAHRE MAXIMUM AUF
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A6		LDX	$41
0001	41			
0002	A9		LDA	#0
0003	00			
0004	D5	MAXM	CMP	$41,X
0005	41			
0006	B0		BCS	NOCHG
0007	02			
0008	B5		LDA	$41,X
0009	41			
000A	CA	NOCHG	DEX	
000B	D0		BNE	MAXM
000C	F7			
000D	85		STA	$40
000E	40			
000F	00		BRK	

Die relative Versetzung für BCS NOCHG ist:

```
 000A
-0008
   02
```

Die relative Versetzung für BNE MAXM ist:

```
 0004     04
       =
-000D    +F3
          F7
```

Die ersten zwei Befehle dieses Programmes bilden den Initialisierungs-Abschnitt.

Dieses Programm benützt die Tatsache, daß Null die kleinste 8-Bit-Binärzahl ohne Vorzeichen ist. Wenn man das Register, das den Maximalwert enthält (in diesem Falle der Akkumulator) auf den kleinstmöglichsten Wert vor dem Eintritt in die Schleife setzt, dann wird das Programm den Akkumulator auf einen größeren Wert setzen, außer alle Elemente in der Anordnung sind Nullen.

Das Programm arbeitet ordnungsgemäß, wenn es zwei Elemente gibt, jedoch nicht, wenn nur eines oder überhaupt keines vorliegt. Warum? Wie könnte man dieses Problem lösen?

Der Befehl CMP $41,X setzt das Übertrags-Flag wie folgt (ELEMENT ist der Inhalt der effektiven Adresse und MAX ist der Inhalt des Akkumulators A):

Übertrag = 0 wenn ELEMENTE > MAX
Übertrag = 1 wenn ELEMENT ⩽ MAX

Erinnern Sie sich daran, daß der Übertrag ein invertiertes Borgen ist. Wenn Übertrag = 1, geht das Programm zur Adresse NOCHG weiter und ändert das Maximum nicht. Wenn Übertrag = 0, ersetzt das Programm das alte Maximum durch das momentane Element durch Ausführung des Befehls LDA $41,X.

Das Programm arbeitet nicht, wenn es sich um Zahlen mit Vorzeichen handelt, da negative Zahlen größer als positive erscheinen werden. Dieses Problem ist etwas verzwickt, da ein Zweierkomplement-Überlauf das Vorzeichen des Ergebnisses verfälschen könnte. Ein weiteres Problem besteht darin, daß der CMP-Befehl das Überlauf-Flag nicht beeinflußt. Ein Programm für Zahlen mit Vorzeichen müßte daher den SBC-Befehl verwenden und sowohl das Vorzeichen wie das Überlauf-Flag prüfen. Das Übertrags-Flag müßte auf 1 vor der Subtraktion gesetzt werden (erinnern Sie sich daran, daß der Übertrag ein invertiertes Borgen ist und der SBC-Befehl es vor der Subtraktion invertiert) und es wäre eine Addition erforderlich, um den ursprünglichen Wert des Maximums wieder herzustellen. Beachten Sie, wie bequem es in diesem Beispiel ist, daß CMP den Inhalt des Akkumulators nicht wirklich ändert.

Justieren einer gebrochenen Binärzahl

Zweck: Verschiebe den Inhalt des Speicherplatzes 0040 nach links, bis das höchstwertige Bit der Zahl 1 ist. Speichere das Ergebnis in den Speicherplatz 0041 und die Anzahl der erforderlichen Links-Verschiebungen in den Speicherplatz 0042. Wenn der Inhalt des Speicherplatzes 0040 null ist, lösche sowohl 0041 wie 0042.

Anmerkung: Dieser Vorgang entspricht der Umwandlung einer Zahl in Exponential-Schreibweise. Zum Beispiel:

$$0.0057 = 5.7 \times 10^{-3}$$

Beispiele:

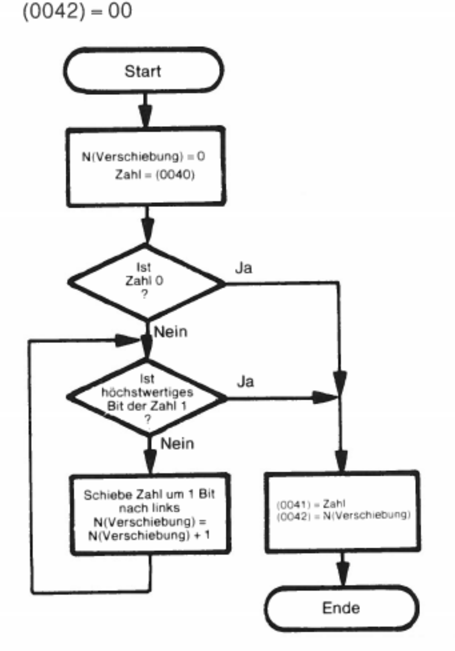
a. (0040) = 22
Ergebnis: (0041) = 88
(0042) = 02

b. (0040) = 01
Ergebnis: (0041) = 80
(0042) = 07

c. (0040) = CB
Ergebnis: (0041) = CB
(0042) = 00

d. (0040) = 00
Ergebnis: (0041) = 00
(0042) = 00

Flußdiagramm:



Quellprogramm:

```
        LDY   #0      ;ANZAHL DER VERSCHIEBUNGEN = 0
        LDA   $40     ;HOLE DATEN
        BEQ   DONE    ;DONE, WENN DATEN NULL
CHKMS   BMI   DONE    ;DONE, WENN MSB = 1
        INY           ;ADDIERE 1 ZUR ANZAHL DER VERSCHIE-
                      ;  BUNGEN
        ASL   A       ;SCHIEBE UM 1 BIT NACH LINKS
        JMP   CHMKS
DONE    STA   $42     ;BEWAHRE JUSTIERTE DATEN AUF
        STY   $42     ;BEWAHRE ANZAHL DER VERSCHIEBUNG-
                      ;  GEN AUF
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
0000	A0		LDY	#0
0001	00			
0002	A5		LDA	$40
0003	40			
0004	F0		BEQ	DONE
0005	07			
0006	30	CHKMS	BMI	DONE
0007	05			
0008	C8		INY	
0009	0A		ASL	A
000A	4C		JMP	CHKMS
000B	06			
000C	00			
000D	85	DONE	STA	$41
000E	41			
000F	84		STY	$42
0010	42			
0011	00		BRK	

BMI DONE bewirkt einen Sprung zum Speicherplatz DONE, wenn das Vorzeichen-Bit 1 ist. Diese Bedingung kann bedeuten, daß das letzte Ergebnis eine negative Zahl war, oder kann einfach bedeuten, daß ihr höchstwertiges Bit 1 war, der Computer liefert nur die Ergebnisse. Der Programmierer muß für die Interpretation sorgen.

ASLA verschiebt den Inhalt des Akkumulators um ein Bit nach links und löscht das niedrigstwertige Bit.

JMP ist ein Befehl für einen unbedingten Sprung, der immer einen neuen Wert in den Befehlszähler plaziert. Er gestattet nur absolute (direkte) oder indirekte Adressierung. Die indirekte Adressierung ergibt höhere Flexibilität, da die tatsächliche Bestimmungsadresse im RAM gespeichert werden kann. Beachten Sie, daß es keine relative Adressierung und keine spezielle Nullseiten-Arten gibt.

Die Adresse im JMP-Befehl wird in zwei aufeinanderfolgenden Speicherplätzen aufbewahrt, mit dem niedrigstwertigen Bit zuerst (bei der niedrigeren Adresse). Dies ist der Standardweg, in dem der Mikroprozessor 6502 erwartet, Adressen aufzufinden, unabhängig davon, ob sie ein Teil von Befehlen sind oder indirekt verwendet werden. Die gleiche Methode von "oben nach unten" wird beim 8080, 8085 und Z80 verwendet, jedoch die entgegengesetzte Lösung (höchstwertge Bits zuerst) wird beim Mikroprozessor 6800 verwendet. Beachten Sie, daß eine Adresse zwei Speicherbytes belegt, obwohl es nur ein einzelnes Datenbyte an dieser Adresse gibt.

Wir könnten dieses Programm neu organisieren, so daß es den zusätzlichen JMP-Befehl eliminiert. Eine neu organisierte Version wäre:

```
        LDY   #0        ;ANZAHL DER VERSCHIEBUNGEN = 0
        LDA   $40       ;HOLE DATEN
        BEQ   DONE      ;DONE, WENN DATEN NULL SIND
CHKMS   INY             ;ADDIERE 1 ZUR ANZAHL DER VERSCHIE-
                        ;  BUNGEN
        ASL   A         ;VERSCHIEBE UM EIN BIT NACH LINKS
        BCC   CHKMS     ;SETZE FORT, WENN MSB NICHT EINS IST
        ROR   A         ;ANDERNFALLS VERSCHIEBE EINMAL ZU-
                        ;  RÜCK
        DEY             ;UND IGNORIERE ZUSÄTZLICHE VERSCHIE-
                        ;  BUNG
DONE    STA   $41       ;BEWAHRE JUSTIERTE DATEN AUF
        STY   $42       ;BEWAHRE ANZAHL DER VERSCHIEBUNG-
                        ;  EN AUF
        BRK
```

Diese Version verschiebt die Daten, bis der Übertrag 1 wird. Es ordnet dann die Daten und die Zahl der Verschiebungen um eins rückwärts an, da die letzte Verschiebung nicht wirklich notwendig war. Zeigen Sie, daß diese Version ebenfalls richtig ist. Was sind ihre Vorteile und Nachteile, verglichen mit dem vorhergehenden Programm? Sie könnten vielleicht einige andere Anordnungen versuchen, um zu sehen, wie sie Ausführungszeit und Speicherbelegung verwenden.

Nach-indizierte (indirekte) Adressierung

NACH-INDIZIERTE (INDIREKTE) ADRESSIERUNG

Wir haben bereits die zusätzliche Flexibilität durch die indizierte Adressierung festgestellt. Die gleichen Befehle können zum Verarbeiten jedes Elementes in einer Anordnung oder Tabelle dienen. Es wird aber auch erhöhte Flexibilität durch die nach-indizierte Adressierung geliefert, bei der der Befehl nur die Adresse auf der Nullseite spezifiziert, die die Basis Adresse der Tabelle oder Anordnung enthält. Nun kann dasselbe Programm eine Anordnung oder Tabelle handhaben, die irgendwo im Speicher liegt. Alles was wir zu tun haben, besteht im Plazieren der Start-Adresse in die entsprechenden Speicherplätze auf der Seite null. Beachten Sie, daß die Start-Adresse zwei Speicherbytes belegt, mit dem niedrigstwertigen Byte zuerst (bei der niedrigeren Adresse). Nach-Indizierung erfordert zusätzliche Takt-Zyklen (sechs vergleichen mit vier für die indizierte Nullseiten-Adressierung) ermöglicht jedoch gewaltige zusätzliche Flexibilität. Ganze Anordnungen müssen nicht mehr bewegt werden, noch sind wiederholte Versionen des gleichen Programmes erforderlich.

Nach-indizierte (indirekte) Adressierung kann nur mit dem Indexregister Y verwendet werden. Daher sieht das Maximum-Programm mit nach-indizierter Adressierung wie folgt aus, wobei angenommen wird, daß die Länge der Anordnung im Speicherplatz 0041 und seine Start-Adresse in den Speicherplätzen 0042 und 0043 liegen.

Beispiel:

```
           (0041) = 05
           (0042) = 43   (LSBs der Start-Adresse minus eins)
           (0043) = 00   (MSBs der Start-Adresse minus eins)
           (0040) = 67   (erstes Element in der Anordnung)
           (0045) = 79
           (0046) = 15
           (0047) = E3
           (0048) = 72
Ergebnis   = (40) = E3, da dies das größte der 5 Zahlen ohne Vorzei-
                  chen ist.
```

Quellprogramm:

```
        LDY   $41        ;HOLE ELEMENT-ZÄHLUNG
        LDA   #0         ;MAXIMUM = NULL (KLEINSTMÖGLICHER
                         ;  WERT)
MAXM    CMP   ($42),Y    ;IST NÄCHSTES ELEMENT ÜBER MAXI-
                         ;  MUM?
        BCS   NOCHG      ;NEIN, BEWAHRE MAXIMUM AUF
        LDA   ($42),Y    ;JA, ERSETZE MAXIMUM DURCH ELEMENT
NOCHG   DEY
        BNE   MAXM       ;SEZTE FORT, BIS ALLE ELEMENTE GE-
                         ;  PRÜFT
        STA   $40        ;BEWAHRE MAXIMUM AUF
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A4		LDY	$41
0001	41			
0002	A9		LDA	#0
0003	00			
0004	D1	MAXM	CMP	($42),Y
0005	42			
0006	B0		BCS	NOCHG
0007	02			
0008	B1		LDA	($42),Y
0009	42			
000A	88	NOCHG	DEY	
000B	D0		BNE	MAXM
000C	F7			
000D	85		STA	$40
000E	40			
000F	00		BRK	

Die indirekte Adresse (in den Speicherplätzen 0042 und 0043) werden auf die gebräuchliche Weise des 6502 gespeichert,mit den niedrigstwertigen Bits zuerst (bei der niedrigeren Adresse).

Wir könnten das gleiche Programm zum Finden des maximalen Elements in einer Anordnung von 5 Eingaben verwenden, beginnend im Speicherplatz 25E1. Alles was wir zu tun haben, besteht im Ändern der indirekten Adresse auf 25E0 vor der Ausführung des Programmes, das heißt:

(0042) = E0 (LSBs der Strat-Adresse minus eins)
(0043) = 25 (MSBs der Start-Adresse minus eins)

Wie würden Sie eine Anordnung handhaben, die im Speicherplatz 25E1 beginnt, wenn das verwendete Programm gewöhnliche indizierte Adressierung (wie in dem früheren Beispiel) verwenden würde? Nehmen Sie an, daß das Programm im ROM liegt, so daß Sie die Adressen in den Befehlen nicht ändern können.

Vor-indizierte (indirekte) Adressierung

VOR-INDIZIERTE (INDIREKTE) ADRESSIERUNG

Die vor-indizierte Adressierung ergibt Ihnen eine andere Art von Flexibilität. Dieses Verfahren gestattet Ihnen die Auswahl einer Adresse aus einer Tabelle von Adressen, anstatt auf eine bestimmte Speicheradresse begrenzt zu sein.
Zum Beispiel könnten wir anstatt des Speicherplatzes 0041, der die Länge der Anordnung in der Maximums-Aufgabe enthält, ihn den Index der Adresse enthalten lassen, die wiederum die Länge der Anordnung enthält. Die Tabelle der Adressen muß irgendwo auf Seite Null liegen, vielleicht beginnend beim Speicherplatz 0060, das heißt

(0060) = 2F
(0061) = 00 } Adresse, in die der Zähler 0 gespeichert ist

(0062) = 80
(0063) = 00 } Adresse, in die der Zähler 1 gespeichert ist

(0064) = A5
(0065) = 00 } Adresse, in der der Zähler 2 gespeichert ist

Ein Problem besteht darin, daß die Adressen zwei Speicherbytes belegen, so daß Sie die Zählerzahl mit zwei multiplizieren müssen, bevor Sie die vor-indizierte Adressierung einsetzen. Beachten Sie, daß alle Adressen in der Art und Weise des 6502 gespeichert werden, mit den niedrigstwertigen Bits zuerst. Vor-indizierte Adressierung ist nicht so nützlich wie nach-indizierte Adressierung, ist jedoch gelegentlich sehr handlich.

AUFGABEN

1) Prüfsumme von Daten

Zweck: Berechne die Prüfsumme einer Serie von Zahlen. Die Länge der Serie befindet sich im Speicherplatz 0041 und die Serie selbst beginnt im Speicherplatz 0042. Speichere die Prüfsumme im Speicherplatz 0040. Die Prüfsumme wird durch Exklusiv-ODERieren aller Zahlen der Serie gebildet.

Anmerkung: Derartige Prüfsummen werden häufig bei Lochstreifen- und Kassetten-Systemen verwendet, um sicherzustellen, daß die Daten ordnungsgemäß gelesen wurden. Die berechnete Prüfsumme wird mit einer anderen verglichen, die mit den Daten gespeichert ist. Wenn die beiden Prüfsummen nicht übereinstimmen, wird das System gewöhnlich entweder dem Operator einen Fehler melden oder die Daten automatisch nochmals lesen.

Beispiel:

(0041) = 03
(0042) = 28
(0043) = 55
(0044) = 26

Ergebnis: (0040) = (0042) ⊕ (0043) ⊕ (0044)
= 28 ⊕ 55 ⊕ 26
= 0 0 1 0 1 0 0 0
⊕ 0 1 0 1 0 1 0 1
0 1 1 1 1 1 0 1
⊕ 0 0 1 0 0 1 1 0
0 1 0 1 1 0 1 1
= 5B

2) Summe von 16-Bit-Daten

Zweck: Berechne die Summe einer Serie von 16-Bit-Zahlen. Die Länge der Serie befindet sich im Speicherplatz 0042 und die Serie selbst beginnt im Speicherplatz 0043. Speichere die Summe in den Speicherplatz 0040 und 0041 (die acht höchstwertigen Bits in 0041). Jede 16-Bit-Zahl belegt zwei Speicherplätze, mit den acht höchstwertigen Bits in der höheren Adresse. Nimm an, daß die Summe in 16 Bits Platz findet.

Beispiel:

(0042) = 03
(0043) = F1
(0044) = 28
(0045) = 1A
(0046) = 30
(0047) = 39
(0048) = 4B

Ergebnis: 28F1 + 301A + 4B89 = A494

(0040) = 94
(0041) = A4

3) Anzahl der Nullen, positiven und negativen Zahlen

Zweck: Bestimme die Anzahl der Nullen, positiven (höchstwertigen Bit null, jedoch die gesamte zahl keine Nullen) und negativen (höchstwertiges Bit 1) Elemente in einem Block. Die Länge des Blocks befindet sich im Speicherplatz 0043 und der Block selbst beginnt in Speicherplatz 0044. Plaziere die Anzahl der negativen Elemente in den Speicherplatz 0040, die Anzahl der Null-Elemente in den Speicherplatz 0041, und die Anzahl der positiven Elemente in den Speicherplatz 0042.

Beispiel:

(0043) = 06
(0044) = 68
(0045) = F2
(0046) = 87
(0047) = 00
(0048) = 59
(0049) = 2A

Ergebnis: 2 negative, 1 Null und 3 positiv, daher
(0040) = 02
(0041) = 01
(0042) = 03

4) Finden des Minimums

Zweck: Finde das kleinste Element in einem Datenblock. Die Länge des Blocks befindet sich im Speicherplatz 0041 und der Block selbst beginnt im Speicherplatz 0042. Speichere das Minimum in den Speicherplatz 0040. Nimm an, daß die Zahlen in dem Block 8-Bit-Binärzahlen ohne Vorzeichen sind.

Beispiel:

(0041) = 05
(0042) = 67
(0043) = 79
(0044) = 15
(0045) = E3
(0046) = 72

Ergebnis: (0040) = 15, da dies die kleinste der fünf Zahlen ohne Vorzeichen ist.

5) Zähle 1-Bits

Zweck: Bestimme, wieviele Bits im Speicherplatz 0040 gleich Eins sind und plaziere das Ergebnis in den Speicherplatz 0041.

Beispiel:

(0040) = 3B = 00111011

Ergebnis: (0041) = 05

Kapitel 6
ZEICHENCODIERTE DATEN

Mikroprozessoren verarbeiten häufig zeichencodierte Daten. Nicht nur Tastaturen, Fernschreiber, Kommunikationsgeräte, Anzeigen und Computer-Terminals erwarten oder liefern zeichencodierte Daten. Auch zahlreiche Instrumente, Test-Systeme und Steuergeräte benötigen Daten in dieser Form. Der am häufigsten verwendete Code ist ASCII. Baudot und EBCDIC findet man seltener. Wir wollen annehmen, daß alle unsere zeichencodierten Daten in Form von 7-Bit-ASCII vorliegen, wobei das höchstwertige Bit 0 ist (siehe Tabelle 6-1).

HANDHABUNG VON DATEN IN ASCII

Einige Grundlagen zur Handhabung von ASCII-codierten Daten sind:

1) **Die Codes für die Zahlen und Buchstaben bilden geordnete Untersequenzen.** Die Codes für Dezimalzahlen sind 30_{16} bis 39_{16}, so daß man eine Umwandlung zwischen dezimal und ASCII mit einem einfachen additiven Faktor durchführen kann. Die Codes für die Großbuchstaben sind 41_{16} bis $5A_{16}$, so daß man eine alphabetische Anordnung durch Sortieren der Daten mit steigender numerischer Ordnung erhalten kann.
2) **Der Computer macht keinen Unterschied zwischen druckenden und nicht druckenden Zeichen.** Diese Unterscheidung wird nur durch die E/A-Bausteine getroffen.
3) **Ein ASCII-Baustein wird nur ASCII-Daten verarbeiten.** Um eine 7 auf einem ASCII-Drucker auszudrucken, muß der Mikroprozessor 37_{16} zum Drucker senden. 07_{16} ist das Zeichen für "Klingel". Ähnlich wird der Mikroprozessor das Zeichen 9 von einer ASCII-Tastatur als 39_{16} empfangen. 09_{16} ist das Tabulator-Zeichen.
4) **Einige ASCII-Bausteine verwenden nicht den vollen Zeichensatz.** Beispielsweise können Steuerzeichen und Kleinbuchstaben ignoriert oder als Zwischenräume oder Fragezeichen gedruckt werden.
5) **Einige häufig verwendete ASCII-Zeichen sind:**

   $0A_{16}$ = line feed (LF = Zeilenvorschub)
   $0D_{16}$ = carriage return (CR = Wagenrücklauf)
   20_{16} = space (Zwischenraum)
   $3F_{16}$ = ? (Fragezeichen)
   $7F_{16}$ = delete character (Löschzeichen)

6) **Jedes ASCII-Zeichen belegt 8 Bits.** Dies gestattet einen größeren Zeichensatz, ist jedoch verschwenderisch, wenn die Daten auf einen kleineren Teilsatz begrenzt sind, wie etwa die Dezimalzahlen. Ein 8-Bit-Byte kann beispielsweise nur eine ASCII-codierte Dezimalzahl enthalten, dagegen jedoch zwei BCD-codierten Ziffern.

Tabelle 6-1. Hex-ASCII-Tabelle.

Hex LSD \ Hex MSD	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

BEISPIELE

Länge einer Zeichenreihe

Zweck: Bestimme die Länge einer Reihe (String oder Kette) von ASCII-Zeichen (7 Bits mit einem höchstwertigen Bit 0). Die Reihe beginnt im Speicherplatz 0041. Das Ende der Reihe wird durch ein Wagenrücklauf-Zeichen ('CR', $0D_{16}$) markiert. Plaziere die Länge der Reihe (ausschließlich des Wagenrücklaufs) im Speicherplatz 0040.

Beispiele:

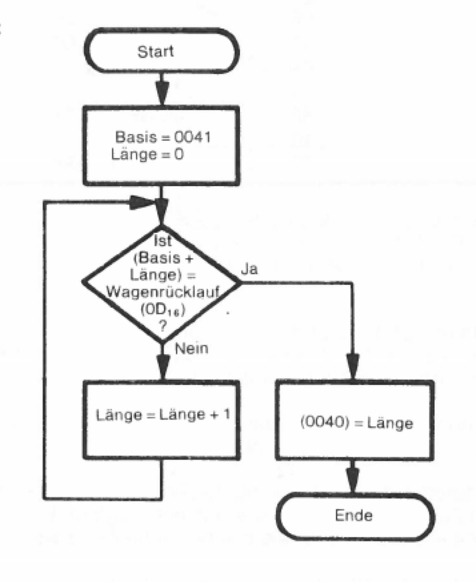
a. (0041) = 0D

Ergebnis: (0040) = 00 da das erste Zeichen ein Wagenrücklauf ist.

b.
(0041) = 52 'R'
(0042) = 41 'A'
(0043) = 54 'T'
(0044) = 48 'H'
(0045) = 45 'E'
(0046) = 52 'R'
(0047) = 0D CR

Ergebnis: (0040) = 06

Flußdiagramm:



Quellprogramm:

```
        LDX   #0      ;LÄNGE DER REIHE = NULL
        LDA   #$0D    ;HOLE WAGENRÜCKLAUF FÜR VERGLEICH
CHKCR   CMP   $41,X   ;IST DAS ZEICHEN EIN WAGENRÜCKLAUF?
        BEQ   DONE    ;JA, DONE
        INX           ;NEIN, ADDIERE 1 ZUR REIHENLÄNGE
        JMP   CHKCR
DONE    STX   $40     ;BEWAHRE REIHENLÄNGE AUF
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
0000	A2		LDX	#0
0001	00			
0002	A9		LDA	#$0D
0003	0D			
0004	D5	CHKCR	CMP	$41,X
0005	41			
0006	F0		BEQ	DONE
0007	04			
0008	E8		INX	
0009	4C		JMP	CHKCR
000A	04			
000B	00			
000C	86	DONE	STX	$40
000D	40			
000E	00		BRK	

Der Wagenrücklauf (CR) ist nur ein weiteres ASCII-Zeichen ($0D_{16}$), soweit es den Computer betrifft. Die Tatsache, daß der Ausgangs-Baustein den Wagenrücklauf als Steuerzeichen anstatt als Druckzeichen behandelt, beeinflußt den Computer nicht.

Der Vergleichs-Befehl CMP setzt die Flags, als ob eine Subtraktion ausgeführt worden wäre, läßt jedoch das Zeichen für den Wagenrücklauf für spätere Vergleiche im Akkumulator. Das Z-Flag wird wie folgt beeinflußt:

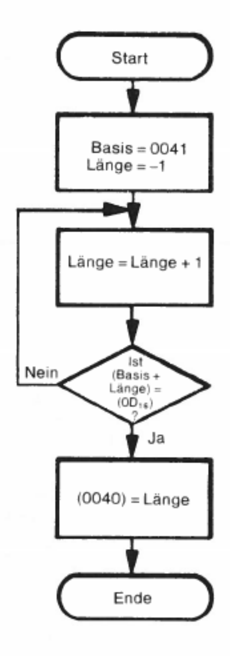
Z = 1 wenn das Zeichen in der Reihe ein Wagenrücklauf ist
Z = 0 wenn es kein Wagenrücklauf ist

Der Befehl INX addiert 1 zum Reihenlängen-Zähler im Index-Register X. LDX#0 initialisiert diesen Zähler auf 0, bevor die Schleife beginnt. Erinnern Sie sich daran, Variable zu initialisieren, bevor sie in einer Schleife verwendet werden.

Diese Schleife begrenzt nicht, da der Zähler nicht auf null dekrementiert wird oder einen Maximalwert erreicht. Der Computer wird einfach mit der Prüfung von Zeichen fortfahren, bis er einen Wagenrücklauf findet. Man muß eine maximale Zählung in eine Schleife wie dieser plazieren, um Probleme mit fehlerhaften Reihen, die keinen Wagenrücklauf enthalten zu vermeiden. Was würde geschehen, wenn das Programmbeispiel mit einer derartigen Reihe verwendet würde?

Beachten Sie, daß man durch Neuanordnung der Logik und Änderung der Anfangsbedingungen dieses Programm verkürzen und die Ausführungszeit verringern kann. Wenn wir das Flußdiagramm so anordnen, daß das Programm den Zähler und den Zeiger inkrementiert, bevor es nach dem Wagenrücklauf Ausschau hält, ist nur eijn Sprungbefehl anstelle von zwei erforderlich. Das neue Flußdiagramm und das Programm sieht dann folgendermaßen aus:

Flußdiagramm:



Quellprogramm:

```
        LDX   #$FF    ;LÄNGE DER REIHE = -1
        LDA   #$0D    ;HOLE ASCII-WAGENRÜCKLAUF FÜR VER-
                      ;  GLEICH
CHKCR   INX           ;ADDIERE 1 ZUR REIHENLÄNGE
        CMP   $41,X   ;IST DAS ZEICHEN EIN WAGENRÜCKLAUF?
        BNE   CHKCR   ;NEIN, PRÜFE NÄCHSTES ZEICHEN
        STX   $40     ;JA, BEWAHRE ZEICHENLÄNGE AUF
        BRK
```

Diese Version ist nicht nur kürzer und schneller, sondern sie beinhaltet auch keine absoluten Ziel-Adressen. Daher kann sie leicht überall in den Speicher plaziert werden. Die frühere Version enthält einen JMP-Befehl mit einer spezifischen absoluten Adresse, während diese Version nur Verzweigungsbefehle mit relativen Adressen besitzt.

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
0000	A2		LDX	#$FF
0001	FF			
0002	A9		LDA	#$0D
0003	0D			
0004	E8	CHKCR	INX	
0005	D5		CMP	$41.X
0006	41			
0007	D0		BNE	CHKCR
0008	FB			
0009	86		STX	$40
000A	40			
000B	00		BRK	

Suchen des ersten Nicht-Zwischenraum-Zeichens

Zweck: Absuchen einer Reihe von ASCII-Zeichen (7 Bits mit höchstwertigem Bit 0) nach einem Nicht-Zwischenraum-Zeichen. Die Reihe beginnt im Speicherplatz 0042. Plaziere den Index des ersten Nicht-Zwischenraum-Zeichens in den Speicherplatz 0040. Ein Zwischenraum-Zeichen ist 20_{16} in ASCII.

Beispiel:

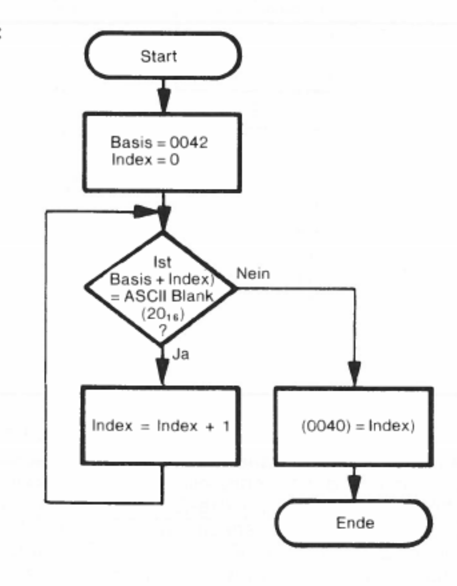
a. (0042) = 37 ASCII

Ergebnis (0040) = 00 da der Speicherplatz 0042 ein Nicht-Zwischenraum-Zeichen enthält.

b.
(0042) = 0 SP
(0043) = 20 SP
(0044) = 20 SP
(0045) = 46 F
(0046) = 20 SP

Ergebnis: (0040) = 03 da die vorausgehenden Speicherplätze lauter Zwischenräume enthalten.

Flußdiagramm:



Quellprogramm:

```
        LDX   #0       ;BEGINNE MIT INDEX = NULL
        LDA   #'       ;HOLE ASCII-ZWISCHEN-RAUM FÜR VER-
                       ;  GLEICH
CHBLK   CMP   $42,X    ;IST DAS ZEICHEN EIN ASCII-
                       ;  ZWISCHENRAUM?
        BNE   DONE     ;NEIN, DONE
        INX            ;JA, PRÜFE NÄCHSTES ZEICHEN
        JMP   CHBLK
DONE    STX   $40      ;BEWAHRE INDEX DES ERSTEN NICHT-
                       ;  ZWISCHENRAUM-ZEICHENS AUF
        BRK
```

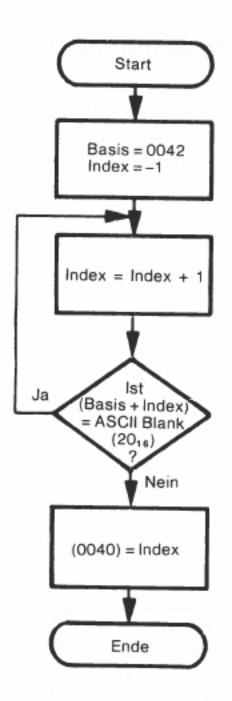
Objektprogramm: Beachten Sie die Verwendung eines Apostrophs (') vor einem ASCII-Zeichen

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
0000	A2		LDX	#0
0001	00			
0002	A9		LDA	#'
0003	20			
0004	D5	CHBLK	CMP	$42.X
0005	42			
0006	D0		BNE	DONE
0007	04			
0008	E8		INX	
0009	4C		JMP	CHBLK
000A	04			
000B	00			
000C	86	DONE	STX	$40
000D	40			
000E	00		BRK	

Das Aufsuchen von Zwischenräumen in Reihen ist eine häufig vorkommende Aufgabe. Zwischenräume werden oft aus Reihen entfernt, wenn sie einfach nur zur Erhöhung der Lesbarkeit oder für spezielle Formate verwendet werden. Es ist eine offensichtliche Verschwendung, Extra-Zwischenräume zu speichern und deren Beginn und Ende zu übertragen, speziell wenn man für die Übertragungszeit und für den Speicherraum zahlen muß. Daten- und Programmeingabe ist jedoch wesentlich einfacher, wenn Extra-Zwischenräume toleriert werden. Mikrocomputer werden häufig in Situationen ähnlich dieser verwendet, um die Form von Daten zwischen von Menschen leicht lesbarer Form und vom Computer und Übertragungsleitungen wirtschaftlich zu handhabenden Formen umzuwandeln.

Wiederum könnten wir, wenn wir die Anfangsbedingungen so ändern, daß der Schleifen-Steuerabschnitt dem Verarbeitungsabschnitt vorausgeht, die Anzahl der Bytes im Programm und die Ausführungszeit der Schleife verringern. Das neu angeordnete Flußdiagramm sieht folgendermaßen aus:

Flußdiagramm:



Quellprogramm:

```
        LDX   #$FF     ;STARTE MIT INDEX = -1
        LDA   #'       ;HOLE ASCII-ZWISCHENRAUM FÜR VER-
                       ;  GLEICH
CHBLK   INX            ;INKREMENTIERE INDEX
        CMP   $42,X    ;IST ZEICHEN EIN ASCII-ZWISCHENRAUM?
        BEQ   CHBLK    ;JA, PRÜFE NÄCHSTES ZEICHEN
        STX   $40      ;NEIN, BEWAHRE INDEX DES ERSTEN
                       ;  NICHT-ZWISCHENRAUM-ZEICHENS AUF
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
0000	A2		LDX	#$FF
0001	FF			
0002	A9		LDA	#'
0003	20			
0004	E8	CHBLK	INX	
0005	D5		CMP	$42.X
0006	42			
0007	F0		BEQ	CHBLK
0008	FB			
0009	86		STX	$40
000A	40			
000B	00		BRK	

Ersetzen führender Nullen durch Zwischenräume

Zweck: Änderung einer Reihe von ASCII-Dezimalzeichen durch Ersetzen aller führender Nullen durch Zwischenräume. Die Reihe beginnt im Speicherplatz 0041. Nimm an, daß sie zur Gänze aus ASCII-codierten Dezimalzahlen besteht. Die Länge der Reihe befindet sich im Speicherplatz 0040.

Beispiele:

a. (0040) = 02
(0041) = 36 ASCII 6

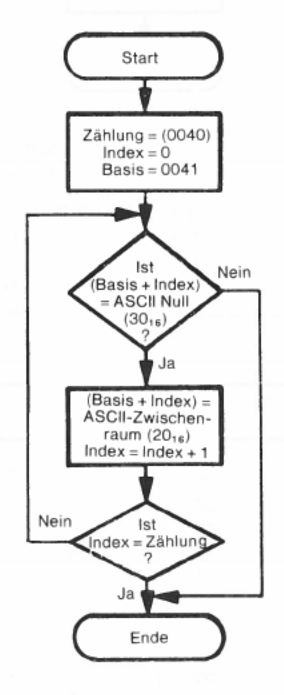
Das Programm läßt die Reihe unverändert, da die führende Ziffer nicht Null ist.

b. (0040) = 08
(0041) = 30 ASCII 0
(0042) = 30 ASCII 0
(0043) = 38 ASCII 8

Ergebnis: (0041) = 20 SP
(0042) = 20 SP

Die beiden führenden ASCII-Nullen wurden durch ASCII-Zwischenräume ersetzt.

Flußdiagramm:



Quellprogramm:

```
        LDX   #0        ;INDEX = NULL FÜR START
        LDY   #'        ;HOLE ASCII-ZWISCHENRUM FÜR ERSATZ
        LDA   #'0       ;HOLE ASCII-NULL FÜR VERGLEICH
CHKZ    CMP   $41,X     ;IST FÜHRENDE ZIFFER NULL
        BNE   DONE      ;NEIN, BEENDE AUSTAUSCH-VORGANG
        STA   $41,X     ;IST FÜHRENDE ZIFFER NULL?
        INX
        CPX   $40
        BNE   CHKZ      ;PRÜFE NÄCHSTE ZIFFER FALLS
                        ;  VORHANDEN
DONE    BRK
```

Ein Apostroph vor einem Zeichen zeigt an, daß der Operand ein ASCII-Code ist.

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A2		LDX	#0
0001	00			
0002	A0		LDY	#'
0003	20			
0004	A9		LDA	#'0
0005	30			
0006	D5	CHKZ	CMP	$41,X
0007	41			
0008	D0		BNE	DONE
0009	07			
000A	94		STY	$41,X
000B	41			
000C	E8		INX	
000D	E4		CPX	$40
000E	40			
000F	D0		BNE	CHKZ
0010	F5			
0011	00	DONE	BRK	

Man wird häufig dezimale Reihen aufbereiten wollen, bevor sie gedruckt oder dargestellt werden, um ihr Aussehen zu verbessern. Häufig vorkommende Aufbereitungs-Aufgaben beinhalten das Eliminieren von führenden Nullen, Überprüfung von Zahlen, Addieren von Vorzeichen oder anderer Identifizierungsmarken und Abrundungen. Offensichtlich können gedruckte Zahlen wie 0006 oder $27.34382 verwirrend und ermüdend sein.

Hier besitzt die Schleife zwei Ausgänge, einen wenn der Prozessor eine Ziffer findet, die nicht Null ist, und den anderen, wenn er die gesamte Reihe überprüft hat.

Der Befehl STY $41,X plaziert ein ASCII-Zwischenraumzeichen (20_{16}) in ein Speicherbyte, das vorher eine ASCII-Null enthielt. Beachten Sie, daß STY nur über eine begrenzte Anzahl von Adressier-Arten verfügt (siehe Tabelle 3-4). Es gibt keine indizierten Adressier-Arten mit dem Register Y, keine Vor-Indizierung und keine absolute Indizierung. Die einzige indizierte Adressier-Art ist die mit der Nullseite zusammen mit Indexregister X.

Es wird angenommen, daß alle Ziffern in einer Reihe in ASCII vorliegen. Das heißt, die Ziffern sind 30_{16} bis 39_{16}, anstatt der gewöhnlichen Dezimalzahlen 0 bis 9. Die Umwandlung von dezimal in ASCII besteht einfach in der Addition von 30_{16} zur Dezimalziffer.

Sie können ein einzelnes ASCII-Zeichen in ein Assemblersprachen-Programm des 6502 plazieren, indem Sie einen Apostroph (') voran stellen. Sie können eine Reihe von ASCII-Zeichen in den Programmspeicher plazieren, indem Sie die Pseudo-Operation .TEXT (Speichere ASCII-Text) mit dem 6502-Assembler verwenden. Ein Begrenzer-Zeichen (gewöhnlich /) muß vor und nach dem Text stehen. Die gebräuchlichste Form ist:

Marke	Code	Operations–Operand
EMSG	.TEXT	/ERROR/

Sie müssen sehr sorgfältig vorgehen, wenn Austast-Nullen vorliegen, und eine Null lassen, im Falle alle Ziffern null sind. Wie würden Sie dies machen?

Beachten Sie, daß jede ASCII-Stelle acht Bits benötigt, verglichen mit vier für eine BCD-Stelle. Deshalb ist ASCII ein aufwendiges Format, wenn man numerische Daten speichert oder überträgt.

Addition gerader Parität zu ASCII-Zeichen

Zweck: Addiere gerade Parität zu einer Reihe von 7-Bit-ASCII-Zeichen. Die Länge der Reihe befindet sich im Speicherplatz 0040, und die Reihe selbst beginnt in Speicherplatz 0041. Plaziere gerade Parität in das höchstwertige Bit jedes Zeichens, indem das höchstwertige Bit auf 1 gesetzt wird, wenn dies für eine Gesamtzahl von 1-Bits im Wort eine gerade Zahl ergibt.

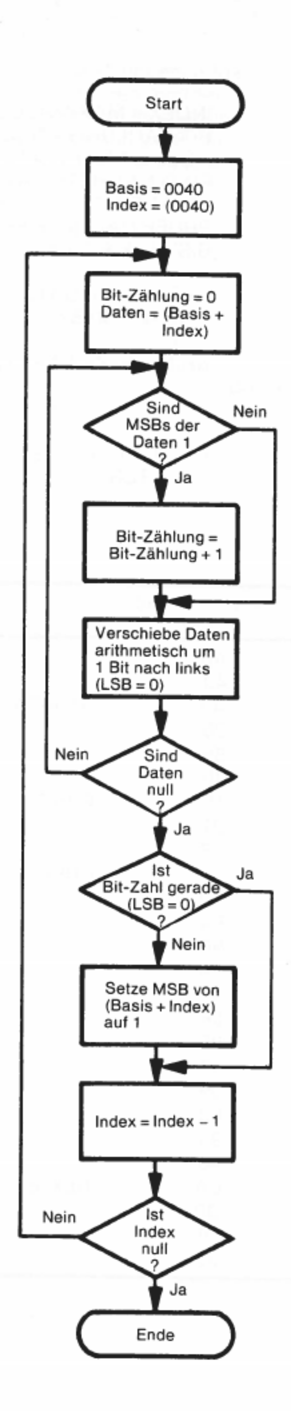
Beispiel:

(0040) = 06
(0041) = 31
(0042) = 32
(0043) = 33
(0044) = 34
(0045) = 35
(0046) = 36

Ergebnis:

(0041) = B1
(0042) = B2
(0043) = 33
(0044) = B4
(0045) = 35
(0046) = 36

Flußdiagramm:



Quellprogramm:

```
        LDX   $40         ;INDEX = MAXIMALE ZÄHLUNG
GTDATA  LDY   #0          ;BIT-ZÄHLUNG = NULL FÜR DATEN
        LDA   $40,X       ;HOLE DATEN VOM BLOCK
CHBIT   BPL   CHKZ        ;SIND NÄCHSTE DATEN BIT 1?
        INY               ;JA, ADDIERE 1 ZUR BIT ZÄHLUNG
CHKZ    ASL   A           ;PRÜFE NÄCHSTE BITPOSITION
        BNE   CHBIT       ;BIS ALLE BITS NULL SIND
        TYA
        LSR   A           ;HATTEN DIE DATEN EINE GERADE ANZAHL
                          ;  VON 1-BITS?
        BCC   NEXTE
        LDA   $40,X       ;NEIN, SETZE PARITÄTSBIT
        ORA   #%10000000
        STA   $40,X
NEXTE   DEX
        BNE   GTDATA      ;MACHE FÜR GESAMTEN DATENBLOCK
                          ;  WEITER
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A6		LDX	$40
0001	40			
0002	A0	GTDATA	LDY	#0
0003	00			
0004	B5		LDA	$40,X
0005	40			
0006	10	CHBIT	BPL	CHKZ
0007	01			
0008	C8		INY	
0009	0A	CHKZ	ASL	A
000A	D0		BNE	CHBIT
000B	FA			
000C	98		TYA	
000D	4A		LSR	A
000E	90		BCC	NEXTE
000F	06			
0010	B5		LDA	$40,X
0011	40			
0012	09		ORA	#%10000000
0013	80			
0014	95		STA	$40,X
0015	40			
0016	CA	NEXTE	DEX	
0017	D0		BNE	GTDATA
0018	E9			
0019	00		BRK	

Die Parität wird häufig zu ASCII-Zeichen addiert, bevor sie auf verrauschten Kommunikationsleitungen übertragen werden, um eine einfache Fehlerprüf-Möglichkeit zu liefern. Die Parität stellt alle Einzelbit-Fehler fest, erlaubt jedoch keine Fehlerkorrektur. Das heißt, man weiß, daß ein Fehler aufgetreten ist, wenn die Parität falsch ist, man kann jedoch nicht angeben, welches Bit verändert wurde. Der Empfänger kann nur eine neuerliche Sendung anfordern.

Das Verfahren zur Berechnung der Parität besteht im Zählen der 1-Bits im Datenwort. Wenn diese Zahl ungerade ist, wird das MSB des Datenwortes auf 1 gesetzt, um die Parität gerade zu machen.

ASL löscht das niedrigstwertige Bit der zu verschiebende Zahl. Deshalb wird das Ergebnis einer Reihe von ASL-Befehlen möglicherweise null sein, unabhängig vom ursprünglichen Wert der Daten (versuchen Sie es!). Der Bitzählungs-Abschnitt des Programmbeispieles benötigt nicht nur keinen Zähler, sondern stoppt auch das Prüfen der Daten, sobald alle verbleibenden Bits Nullen sind. Dieses Verfahren spart in zahlreichen Fällen Ausführungszeit.

Das MSB der Daten wird auf 1 gesetzt, indem es logisch mit einem Muster ODERiert wird, das eine 1 in seinem höchstwertigen Bit und sonst Nullen besitzt. Das logische ODERieren eines Bits mit 1 erzeugt ein Ergebnis von 1, unabhängig vom ursprünglichen Wert, während das logische ODERieren eines Bits mit 0 den ursprünglichen Wert nicht ändert.

Übereinstimmung von Bitmustern

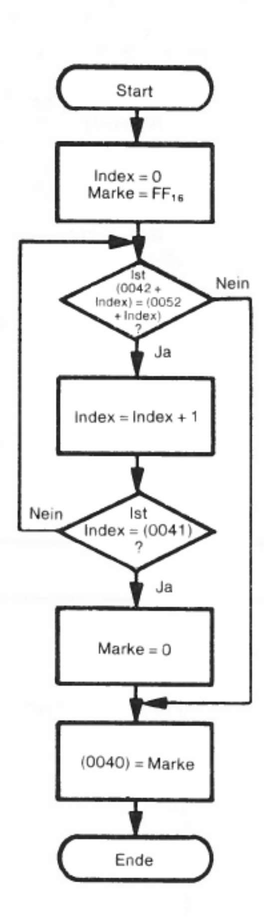
Zweck: Vergleiche zwei Reihen von ASCII-Zeichen, um festzustellen, ob sie gleich sind. Die Länge der Reihen befindet sich im Speicherplatz 0041. Eine Reihe beginnt im Speicherplatz 0042 und die andere im Speicherplatz 0052. Wenn die beiden Reihen übereinstimmen, lösche Speicherplatz 0040. Andernfalls setze Speicherplatz 0040 auf FF_{16} (alles Einsen).

Beispiele:

a.	(0041)	=	03	
	(0042)	=	43	'C'
	(0043)	=	41	'A'
	(0044)	=	54	'T'
	(0052)	=	43	'C'
	(0053)	=	41	'A'
	(0054)	=	54	'T'
Ergebnis:	(0040)	=	00 da die beiden Reihen gleich sind.	
b.	(0041)	=	03	
	(0042)	=	52	'R'
	(0043)	=	41	'A'
	(0044)	=	54	'T'
	(0052)	=	43	'C'
	(0053)	=	41	'A'
	(0054)	=	54	'T'
Ergebnis:	(0040)	=	FF da sich die ersten Zeichen in der Reihe unterscheiden	

Anmerkung: Der Vorgang für das Suchen nach Übereinstimmung endet, sobald die CPU einen Unterschied findet. Der Rest der Reihen muß nicht überprüft werden.

Flußdiagramm:



Quellprogramm:

```
        LDX   #0       ;STARTE MIT ERSTEM ELEMENT IN DEN-
                       ;  REIHEN
        LDY   #$FF     ;MARKIERER FÜR KEINE ÜBEREIN-
                       ;  STIMMUNG
CHCAR   LDA   $42,X    ;HOLE ZEICHEN VON REIHE 1
        CMP   $52,X    ;GIBT ES EINE ÜBEREINSTIMMUNG MIT
                       ;  REIHE 2?
        BNE   DONE     ;NEIN, DONE
        INX
        CPX   $41
        BNE   CHCAR    ;PRÜFE NÄCHSTES PAAR, WENN NOCH
                       ;  VORHANDEN
        LDY   #0       ;WENN KEINES VORHANDEN, MARKIERE
                       ;  ÜBEREINSTIMMUNG
DONE    STY   $40      ;BEWAHRE ÜBEREINSTIMMUNGS-
                       ;  MARKIERUNG AUF
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A2		LDX	#0
0001	00			
0002	A0		LDY	#$FF
0003	FF			
0004	B5	CHCAR	LDA	$42,X
0005	42			
0006	D5		CMP	$52,X
0007	52			
0008	D0		BNE	DONE
0009	07			
000A	E8		INX	
000B	E4		CPX	$41
000C	41			
000D	D0		BNE	CHCAR
000E	F5			
000F	A0		LDY	#0
0010	00			
0011	84	DONE	STY	$40
0012	40			
0013	00		BRK	

Die Feststellung der Übereinstimmung von Reihen aus ASCII-Zeichen ist ein wesentlicher Teil für das Erkennen von Namen und Kommandos, Identifizierung von Variablen oder Operationscodes in Assemblern und Kompilierern, Finden von Dateien und zahlreicher anderer Aufgaben.

Das Indexregister X wird zum Zugriff auf beide Reihen verwendet, nur die Basis-Adressen sind verschieden. Dieses Verfahren gestattet uns, die Reihen überall im Speicher zu plazieren, obwohl wir absolute indizierte Adressierung verwenden müßten, wenn die Reihen nicht auf der Nullseite liegen. Wir könnten auch die nach-indizierte Adressier-Art (mit Indexregister Y) verwenden, wenn wir zwei verschiedene Basis-Adressen irgendwo auf Seite Null gespeichert hätten.

Der Befehl CMP $52,X vergleicht den Akkumulator mit dem Inhalt des indizierten Speicherplatzes. Wir könnten den Befehl LDY #0 durch INY ersetzen. Weshalb? Vergleichen Sie die Zeit und die Speicheranforderungen der beiden Lösungen. Welche Version würden Sie für deutlicher halten? Dieser Austausch würde Ihnen auch gestatten, einen Speicherplatz für den Markierer zu verwenden, anstatt eines Registers (weshalb?).

AUFGABEN

1) Länge einer Fernschreib-Nachricht

Zweck: Bestimme die Länge einer ASCII-Nachricht. Alle Zeichen sind 7-Bit-ASCII mit MSB = 0. Die Reihe von Zeichen, in der die Nachricht eingebettet ist, beginnt im Speicherplatz 0041. Die Nachricht selbst beginnt mit einem ASCII-STX-Zeichen (02_{16}) und endet mit ETX (03_{16}). Plaziere die Länge der Nachricht (die Anzahl der Zeichen zwischen dem STX und ETX, jedoch ausschließlich der beiden) in den Speicherplatz 0040.

Beispiel:

	(0041)	=	40	
	(0042)	=	02	STX
	(0043)	=	47	'G'
	(0044)	=	4F	'O'
	(0054)	=	03	ETX
Ergebnis:	(0040)	=	02, da es zwei gleiche Zeichen zwischen dem STX im Speicherplatz 0042 und ETX im Speicherplatz 0045 gibt.	

2) Suchen des letzten Nicht-Zwischenraum-Zeichens

Zweck: Absuchen einer Reihe von ASCII-Zeichen nach dem letzten Nicht-Zwischenraum-Zeichen. Die Reihe beginnt im Speicherplatz 0042 und endet mit einem Wagenrücklauf-Zeichen ($0D_{16}$). Plaziere die Adresse des letzten Nicht-Zwischenraum-Zeichens in den Speicherplatz 0040.

Beispiele:

a.	(0042)	=	37	ASCII 7
	(0043)	=	0D	CR CR
Ergebnis:	(0040)	=	00, da das letzte (und einzige) Nicht-Zwischenraum-Zeichen im Speicherplatz 0042 liegt.	
b.	(0042)	=	41	'A'
	(0043)	=	20	SP
	(0044)	=	48	'H'
	(0045)	=	41	'A'
	(0046)	=	54	'T'
	(0047)	=	20	SP
	(0048)	=	20	SP
	(0049)	=	0D	CR
Ergebnis	(0040)	=	04	

3) Abrunden von dezimalen Reihen in ganzzahlige Form

Zweck: Aufbereiten einer Reihe von ASCII-Dezimalzeichen, indem alle Ziffern rechts vom Dezimalpunkt durch ASCII-Zwischenräume (20_{16}) ersetzt werden. Die Reihe beginnt im Speicherplatz 0041, und es wird angenommen, daß sie zur Gänze aus ASCII-codierten Dezimalziffern sowie einem möglichen Dezimalpunkt ($2E_{16}$) bestehen. Die Länge der Reihe befindet sich im Speicherplatz 0040. Wenn kein Dezimalpunkt in der Reihe auftritt, so kann angenommen werden, daß der Dezimalpunkt ganz am rechten Ende liegt.

Beispiel:

a.	(0040)	=	04	
	(0041)	=	37	ASCII 7
	(0042)	=	2E	ASCII
	(0043)	=	38	ASCII 8
	(0044)	=	31	ASCII 1
Ergebnis:	(0041)	=	37	ASCII 7
	(0042)	=	2E	ASCII
	(0043)	=	20	SP
	(0044)	=	20	SP
b.	(0040)	=	03	
	(0041)	=	26	ASCII 6
	(0042)	=	37	ASCII 7
	(0043)	=		ASCII 1
Ergebnis:	Unverändert, da die Zahl mit 671 angenommen wurde.			

4) Prüfen auf gerade Parität in ASCII-Zeichen

Zweck: Prüfen auf gerade Parität in einer Reihe von ASCII-Zeichen. Die Länge der Reihe liegt im Speicherplatz 0041 und die Reihe selbst beginnt im Speicherplatz 0042. Wenn die Parität korrekt ist, lösche Speicherplatz 0040. Andernfalls plaziere FF_{16} (alles Einsen) in den Speicherplatz 0040.

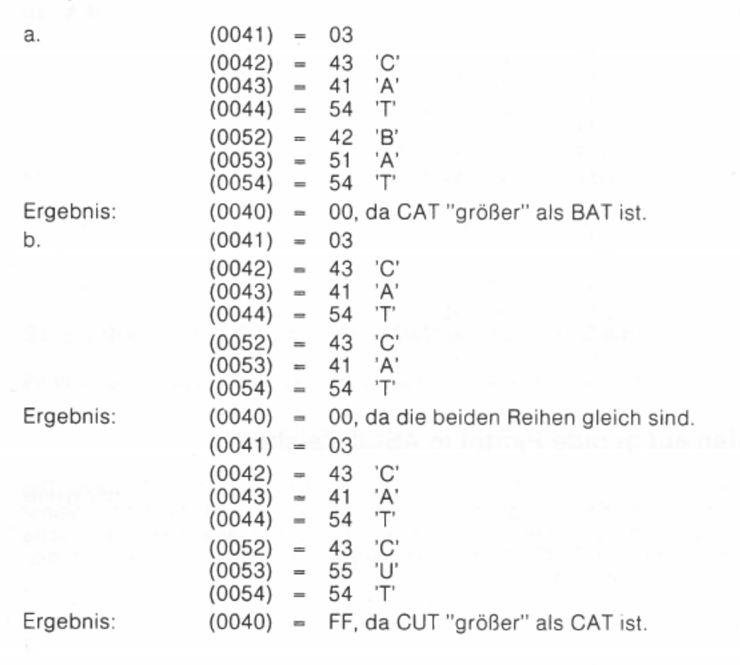
Beispiele:

a.	(0041)	=	03
	(0042)	=	B1
	(0043)	=	B2
	(0044)	=	33
Ergebnis	(0040)	=	00, da alle Zeichen gerade Parität haben.
b.	(0041)	=	03
	(0042)	=	B1
	(0043)	=	B6
	(0044)	=	33
Ergebnis	(0040)	=	FF, da die Zeichen im Speicherplatz 0043 nicht gerade Parität besitzen.

5) Vergleich von Reihen

Zweck: Vergleiche zwei Reihen von ASCII-Zeichen, um festzustellen, welche größer ist (das heißt welche der anderen in "alphabetischer" Reihenfolge folgt). Die Länge der Reihen befindet sich im Speicherplatz 0041. Eine Reihe beginnt im Speicherplatz 0042 und die andere im Speicherplatz 0052. Wenn die Reihe, die im Speicherplatz 0042 beginnt, größer oder gleich wie die andere Reihe ist, lösche Speicherplatz 0040. Andernfalls setze Speicherplatz 0040 auf FF_{16} (alles Einsen).

Beispiele:



a.	(0041)	= 03	
	(0042)	= 43	'C'
	(0043)	= 41	'A'
	(0044)	= 54	'T'
	(0052)	= 42	'B'
	(0053)	= 51	'A'
	(0054)	= 54	'T'
Ergebnis:	(0040)	= 00, da CAT "größer" als BAT ist.	
b.	(0041)	= 03	
	(0042)	= 43	'C'
	(0043)	= 41	'A'
	(0044)	= 54	'T'
	(0052)	= 43	'C'
	(0053)	= 41	'A'
	(0054)	= 54	'T'
Ergebnis:	(0040)	= 00, da die beiden Reihen gleich sind.	
	(0041)	= 03	
	(0042)	= 43	'C'
	(0043)	= 41	'A'
	(0044)	= 54	'T'
	(0052)	= 43	'C'
	(0053)	= 55	'U'
	(0054)	= 54	'T'
Ergebnis:	(0040)	= FF, da CUT "größer" als CAT ist.	

Kapitel 7
CODE-UMWANDLUNG

Code-Umwandlung ist eine ständig vorkommende Aufgabe für Mikroprozessoren. Periphere Geräte liefern Daten in ASCII, BCD oder zahlreichen speziellen Codes. Der Computer muß diese Daten in binär oder dezimal umwandeln, um sie weiter verarbeiten zu können. Ausgangs-Bausteine benötigen Daten in ASCII, BCD, Sieben-Segment oder in anderen Codes. Daher muß der Computer die Ergebnisse in eine geeignete Form umwandeln, nachdem die Verarbeitung beendet ist.

Es gibt verschiedene Möglichkeiten zur Lösung einer Code-Umwandlung:

1) **Einige Umwandlungen können leicht durch Algorithmen gehandhabt werden, die arithmetische oder logische Funktionen beinhalten.** Das Programm muß jedoch möglicherweise einige spezielle Fälle seperat verarbeiten.
2) **Komplexere Umwandlungen können mit Nachschlagetabellen ausgeführt werden.** Das Verfahren mit den Nachschlagetabellen benötigt wenig Programmierung und ist leicht anzuwenden. Die Tabelle kann jedoch einen großen Teil des Speichers belegen, wenn der Bereich der Eingangswerte sehr umfangreich ist.
3) **Für einige Umwandlungs-Aufgaben gibt es leicht verfügbare Hardware.** Typische Beispiele sind Decoder für BCD-zu-7-Segment-Umwandlung und universelle asynchrone Empfänger/Sender (UARTs) für die Umwandlung zwischen parallelen (ASCII) und seriellen (Fernschreiber-) Formaten.

In den meisten Anwendungen sollte das Programm soviel wie möglich von der Code-Umwandlung ausführen. Dies ergibt eine Ersparnis an Bauteilen und Platz auf der Printplatte, sowie erhöhte Zuverlässigkeit. Ferner sind die meisten Code-Umwandlungen leicht zu programmieren und benötigen wenig Ausführungszeit.

BEISPIELE

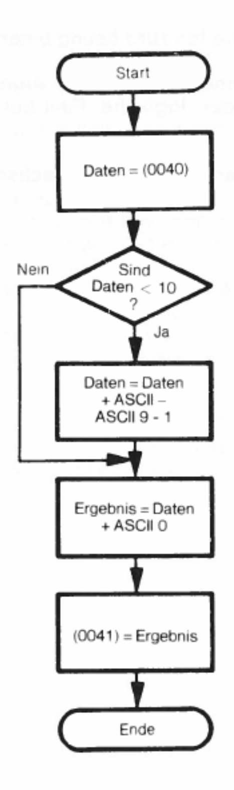
Hexadezimal in ASCII

Zweck: Wandle den Inhalt des Speicherplatzes 0040 in ein ASCII-Zeichen um. Der Speicherplatz 0040 enthält eine einzelne hexadezimale Ziffer (die vier höchstwertigen Bits sind null). Speichere das ASCII-Zeichen in den Speicherplatz 0041.

Beispiele:

a.	(0040)	=	0C	
Ergebnis:	(0041)	=	43	'C'
b.	(0040)	=	06	
Ergebnis:	(0041)	=	36	'6'

Flußdiagramm:



Quellprogramm:

```
        LDA   $40         ;HOLE DATEN
        CMP   #10         ;SIND DATEN KLEINER ALS 10?
        BCC   ASCZ
        ADC   #'A-'9-1    ;NEIN ADDIERE VERSETZUNG FÜR BUCH-
                          ;   STABEN (ÜBERTRAG = 1)
ASCZ    ADC   #'0         ;ADDIERE VERSETZUNG FÜR ASCII
        STA   $41         ;SPEICHERE ASCII-ZIFFER
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A5		LDA	$40
0001	40			
0002	C9		CMP	#10
0003	0A			
0004	90		BCC	ASCZ
0005	02			
0006	69		ADC	#'A-'9-2
0007	06			
0008	69	ASCZ	ADC	#'0
0009	30			
000A	85		STA	$41
000B	41			
000C	00		BRK	

Der grundlegende Gedanke in diesem Programm besteht in der Addition von ASCII Null (30_{16}) zu allen hexadezimalen Ziffern. Diese Addition wandelt die Dezimalziffer korrekt in ASCII um. Es besteht jedoch ein Unterschied zwischen ASCII 9 (39_{16}) und ASCII A (41_{16}), der berücksichtigt werden muß. Dieser Unterschied muß zu den nicht dezimalen Ziffern A, B, C, D, E und F addiert werden. Dies wird durch den ersten ADC-Befehl durchgeführt, der die Versetzung 'A-'9-2 zum Inhalt des Akkumulators addiert. Können Sie erklären, weshalb die Versetzung gleich 'A-'9-2 ist?

Beachten Sie, daß die Additions-Faktoren in das Assembler-Sprachen-Programm in ASCII-Form plaziert werden (ein einzelnes Fragezeichen oder Apostroph geht einem ASCII-Zeichen voraus). Die Versetzung für die Buchstaben wird als arithmetischer Ausdruck belassen. Der Zweck der Faktoren wird hierbei in der Auflistung der Assemblersprache so deutlich wie möglich gemacht. Die zusätzliche Assemblierzeit ist sehr klein im Vergleich zum Gewinn an Deutlichkeit.

Erinnern Sie sich daran, daß der ADC-Befehl immer das Übertragsbit addiert. Nach dem BCC-Befehl wissen wir, daß der Übertrag eins enthält (andernfalls würde eine Verzweigung aufgetreten sein). So reduzieren wir einfach den additiven Faktor um 1 um den Übetrag zu berücksichtigen. Beim zweiten ADC-Befehl wird der Übertrag null sein, wenn das Programm nach dem CMP-Befehl verzweigte (da der BCC-Befehl verwendet wurde), oder der Akkumulator enthielt eine gültige Hexadezimalzahl (10 bis 15), da der additive Faktor nur 7 ist. Daher brauchen wir uns im allgemeinen nicht um den Übertrag zu kümmern.

Diese Routine kann in einer Vielzahl von Programmen verwendet werden. Zum Beispiel müssen Monitorprogramme hexadezimale Ziffern in ASCII umwandeln, um den Inhalt von Speicherplätzen in hexadezimal auf einem ASCII-Drucker oder auf einem Bildschirm anzuzeigen.

Eine andere (raschere) Umwandlungs-Methode, die überhaupt keine bedingten Sprünge benötigt, stellt das folgende Programm dar, das von Allison[1] beschrieben wurde.

```
SED             ;MACHE ADDITION DEZIMAL
      CLC       ;LÖSCHE ÜBERTRAG ZU BEGINN
LDA   $40       ;HOLE HEXADEZIMALZAHL
ADC   #$90      ;ENTWICKLE ZUSÄTZLICHE 6 UND ÜBER-
                ;  TRAG
ADC   #$40      ;ADDIERE ASCII OFFSET IN ÜBERTRAG
STA   $41       ;SPEICHERE ASCII-ZIFFER
CLD             ;LÖSCHE DEZIMAL-BETRIEBSART VOR DEM
                ;  ABSCHLUSS
BRK
```

Versuchen Sie das Programm mit einigen Ziffern. Können Sie erklären wie es arbeitet? Beachten Sie, daß Sie das Flag für die Dezimalbetriebsart sorgfältig löschen müssen, wenn Sie alle dezimalen arithmetischen Vorgänge beendet haben. Andernfalls würden Sie dezimale Ergebnisse in Programmen erhalten (einschließlich des Monitors), wo sie nicht erwünscht sind.

Dezimal in Sieben-Segment

Zweck: Wandle den Inhalt des Speicherplatzes 0041 in einen Sieben-Segment-Code im Speicherplatz 0042 um. Wenn der Speicherplatz 0041 keine einzelne Dezimalziffer enthält, lösche Speicherplatz 0042.

Sieben-Segment-Tabelle: Die folgende Tabelle kann zur Umwandlung von Dezimalzahlen in den Sieben-Segment-Code verwendet werden. Der Sieben-Segment-Code ist so organisiert, daß das höchstwertige Bit immer null ist, gefolgt vom Code (1 = ein, 0 = aus) für die Segmente g, f, e, d, c, b und a (siehe Bild 7-1).

Die Segment-Bezeichnungen sind standardisiert, die von uns gewählte Organisation ist jedoch willkürlich. In tatsächlichen Anwendungen bestimmt die Hardware die Zuweisung von Datenbits zu den Segmenten.

Beachten Sie, daß die Tabelle 7D für 6 anstatt 7C (oberer Querbalken ausgeschaltet) verwendet, um Verwechslung mit dem kleinen b zu vermeiden, und 6F für 9 anstatt 67 (unterer Querbalken aus), jedoch aus keinem bestimmten Grund.

Ziffer	Code
0	3F
1	06
2	5B
3	4F
4	66
5	6D
6	7D
7	07
8	7F
9	6F

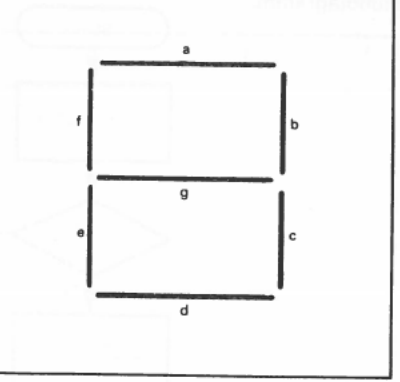
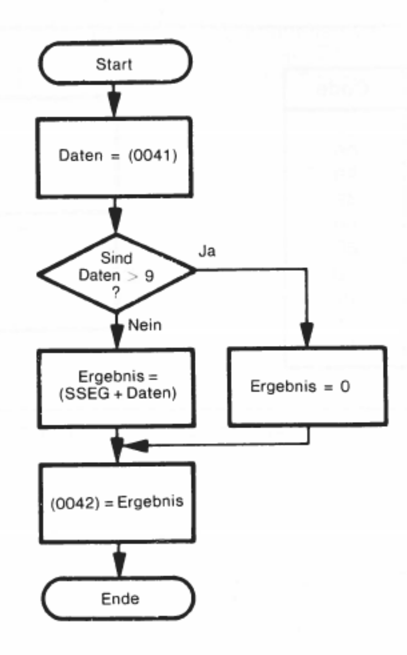


Bild 7-1. Sieben-Segment-Anordnung.

Beispiel:

a.	(0041) =	03
Ergebnis:	(0042) =	4F
b.	(0041) =	28
Ergebnis:	(0042) =	00

Flußdiagramm:



Beachten Sie, daß die Addition der Basis-Adresse (SSEG) mit dem Index (DATA) die Adresse ergibt, welche die Antwort enthält.

Quellprogramm:

```
        LDA     #0          ;HOLE FEHLER-CODE ZUM AUSTASTEN
                            ;  DER ANZEIGE
        LDX     $41         ;HOLE DATEN
        CPX     #10         ;SIND DATEN EINE DEZIMALZIFFER?
        BCS     DONE        ;NEIN, BEWAHRE FEHLERCODE AUF
        LDA     SSEG,X      ;JA, HOLE SIEBEN -SEGMENTCODE AUS
                            ;  TABELLE
DONE    STA     $42         ;BEWAHRE SIEBEN-SEGMENTCODE ODER
                            ;  FEHLERCODE AUF
                BRK
        *=$20               ;SIEBEN-SEGMENT-CODETABELLE
        SSEG    .BYTE       $3F, $06, $5B, $4F, $66
                .BYTE       $6D, $7D, $07, $7F, $6F
```


Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
0000	A9		LDA	#0
0001	00			
0002	A6		LDX	$41
0003	41			
0004	E0		CPX	#10
0005	0A			
0006	B0		BCS	DONE
0007	02			
0008	B5		LDA	SSEG,X
0009	20			
000A	85	DONE	STA	$42
000B	42			
000C	00		BRK	
0020	3F	SSEG	.BYTE	$3F
0021	06			$06
0022	5B			$5B
0023	4F			$4F
0024	66			$66
0025	6D		.BYTE	$6D
0026	7D			$7D
0027	07			$07
0028	7F			$7F
0029	6F			$6F

Das Programm berechnet die Speicher-Adresse des gewünschten Codes durch Addition des Index (d.h. die darzustellende Ziffer) zur Basis-Adresse der 7-Segment-Codetabelle. Das Verfahren ist als Nachschlagen in einer Tabelle bekannt. Es sind keine besonderen Befehle für die Addition erforderlich, da sie automatisch in den indizierten Adressier-Arten ausgeführt werden.

Die Pseudo-Operation .BYTE der Assemblersprache plaziert konstante Daten in den Programmspeicher. Derartige Daten können Tabellen, Nachrichten-Köpfe, Fehlermeldungen, Formatzeichen, Schwellen etc. beinhalten. Die Marke, die zu einer Pseudo-Operation .BYTE gehört, wird dem Wert der Adresse zugewiesen, in die das erste Datenbyte plaziert wird.

Für Code-Umwandlungen werden häufig Tabellen verwendet, die wesentlich komplexer als das vorhergehende Beispiel sind. Derartige Tabellen enthalten typisch alle Ergebnisse, die entsprechend der Eingangsdaten organisiert sind (die erste Eingabe ist der Code, der der Zahl Null entspricht).

Sieben-Segment-Anzeigen liefern gut lesbare Formen für die Dezimalziffern und einige wenige Buchstaben sowie andere Zeichen. Rechnerartige Sieben-Segment-Anzeigen sind preisgünstig, leicht zu kombinieren und benötigen wenig Leistung. Die sieben-segment-codierten Ziffern sind jedoch etwas schwierig zu lesen.

Der Assembler plaziert einfach die Daten für die Tabelle in den Speicher. Beachten Sie, daß eine einzelne Pseudo-Operation .BYTE zahlreiche Speicherplätze füllen kann.

Wir haben etwas Speicherplatz zwischen dem Programm und der Tabelle freigelassen, um spätere Ergänzungen oder Korrekturen einfügen zu können.

Die Tabelle kann überall in den Speicher plaziert werden, obwohl die absolute indizierte Adressier-Art zu verwenden wäre, wenn sie nicht auf der Nullseite liegt. Wir könnten auch Nach-Indizierung (mit Indexregister Y) verwenden und die Basis-Adresse in zwei Speicherbytes auf der Nullseite aufbewahren. Das gleiche Programm könnte dann mit jeder beliebigen Tabelle verwendet werden, da die Basis-Adresse im RAM anstatt im ROM spezifiziert würde.

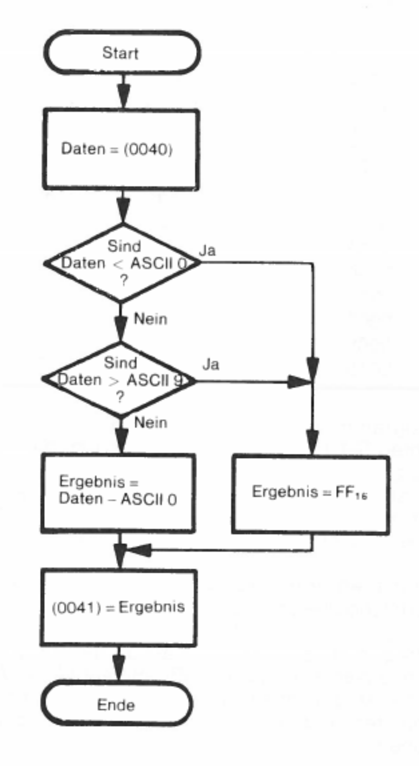
ASCII in dezimal

Zweck: Wandle den Inhalt des Speicherplatzes 0040 von einem ASCII-Zeichen in eine Dezimalziffer um und speichere das Ergebnis in den Speicherplatz 0041. Wenn der Inhalt des Speicherplatzes 0040 nicht die ASCII-Darstellung einer Dezimalziffer ist, setze den Inhalt des Speicherplatzes 0041 auf FF_{16}.

Beispiele:

a.	(0040)	= 37 (ASCII 7)
Ergebnis:	(0041)	= 07
b.	(0040)	= 55
Ergebnis:	(0041)	= FF (ein ungültiger Code, da er keine ASCII-Dezimalziffer ist).

Flußdiagramm:



Quellprogramm:

```
        LDX   #$F      ;HOLE FEHLER-MITTEILUNG
        LDA   $40      ;HOLE DATEN
        SEC            ;IGNORIERE ÜBERTRAG BEI SUBTRAKTION
        SBC   #'0      ;SIND DATEN UNTER ASCII-NULL?
        BCC   DONE     ;JA, KEINE ZIFFER
        CMP   #10      ;SIND DATEN ÜBER ASCII-NEUN?
        BCS   DONE     ;JA, KEINE ZIFFER
        TAX            ;BEWAHRE ZIFFER AUF FALLS GÜLTIG
DONE    STX   $41      ;BEWAHRE ZIFFER ODER FEHLERMARKIE-
                       ;  RUNG AUF
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A2		LDX	#$FF
0001	FF			
0002	A5		LDA	$40
0003	40			
0004	38		SEC	
0005	E9		SBC	#'0
0006	30			
0007	90		BCC	DONE
0008	05			
0009	C9		CMP	#10
000A	0A			
000B	B0		BCS	DONE
000C	01			
000D	AA		TAX	
000E	86	DONE	STX	$41
000F	41			
0010	00		BRK	

Dieses Programm verarbeitet ASCII-codierte Zeichen so wie gewöhnliche Zahlen. Beachten Sie, daß die Dezimalziffern und die Buchstaben Gruppen von aufeinanderfolgenden Codes bilden. Reihen (Strings) von Buchstaben (wie Namen) können in alphabetischer Reihenfolge geordnet werden, indem ihre ASCII-Darstellungen mit steigender numerischer Ordnung (ASCII B = ASCII A + 1, beispielsweise) plaziert werden.

Das Subtrahieren von ASCII 0 (30_{16}) von einer ASCII-Dezimalziffer ergibt die BCD-Darstellung dieser Ziffer.

Der Übertrag muß vor einer Subtraktion gesetzt werden, wenn er das Ergebnis nicht beeinflussen soll, da der SBC-Befehl (A) = (A) – (M) – (1 – Übertrag) erzeugt, wobei M der Inhalt des adressierten Speicherplatzes ist. Vergleichsbefehle beinhalten andererseits nicht den Übertrag in den in ihnen enthaltenen Subtraktionen.

Eine Umwandlung ASCII in dezimal ist erforderlich, wenn Dezimalzahlen von einem ASCII-Gerät, wie einem Fernschreiber oder einem Bildschirm-Terminal, eingegeben werden.

Der grundlegende Gedanke dieses Programmes besteht in der Feststellung, ob das Zeichen zwischen ASCII 0 und ASCII 9 (einschließlich) liegt. Wenn dies der Fall ist, so ist es eine ASCII-Dezimalziffer, da die Ziffern eine Sequenz bilden. Sie kann dann einfach in dezimal umgewandelt werden, indem 30_{16} (ASCII 0) subtrahiert wird. Zum Beispiel: ASCII 7 –ASCII 0 = 7 – 0 = 7.

Beachten Sie, daß ein Vergleich mittels einer tatsächlichen Subtraktion (SBC#'0) ausgeführt wird, da die Subtraktion zur Umwandlung von ASCII in dezimal erforderlich ist. Der andere Vergleich wird mittles einer implizierten Subtraktion (CMP#10) durchgeführt, da das endgültige Ergebnis nun im Akkumulator liegt, wenn die Ursprungszahl gültig ist.

BCD in binär

Zweck: Wandle zwei BCD-Stellen in den Speicherplätzen 0040 und 0041 in eine Binärzahl im Speicherplatz 0042 um. Die höchstwertige BCD-Stelle liegt im Speicherplatz 0040.

Beispiele:

a. (0040) = 02
(0041) = 09

Ergebnis: (0042) = $1D_{16}$ = 29_{10}

b. (0040) = 07
(0041) = 01

Ergebnis: (0042) = 47_{16} = 71_{10}

Anmerkung: Es wird kein Flußdiagramm angegeben, da das Programm die höchstwertige Stelle mit 10 multipliziert, indem es einfach die Formel $10x = 8x + 2x$ verwendet. Die Multiplikation mit 2 erfordert eine arithmetische Links-Verschiebung und die Multiplikation mit 8 benötigt drei derartige Verschiebungen.

Quellprogramm:

```
LDA   $40     ;HOLE HÖCHSTWERTIGE STELLE (MSD)
ASL   A       ;MULTIPLIZIERE MSD MIT 2
STA   $42     ;BEWAHRE DOPPELTES MSD AUF
ASL   A       ;MSD MAL 4
ASL   A       ;MSD MAL 8
CLC
ADC   $42     ;MSD MAL 10 (KEIN ÜBERTRAG)
ADC   $41     ;ADDIERE NIEDRIGSTWERTIGE STELLE
STA   $42     ;SPEICHERE BINÄRES ÄQUIVALENT
BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)	
0000	A5	LDA	$40
0001	40		
0002	0A	ASL	A
0003	85	STA	$42
0004	42		
0005	0A	ASL	A
0006	0A	ASL	A
0007	18	CLC	
0008	65	ADC	$42
0009	42		
000A	65	ADC	$41
000B	41		
000C	85	STA	$42
000D	42		
000E	00	BRK	

BCD-Eingaben werden in binär umgewandelt, um Speicherplatz zu sparen und Berechnungen zu vereinfachen. Die Umwandlung kann jedoch einige Vorteile der binären Speicherung und Arithmetik aufheben.

Dieses Programm multipliziert die BCD-Zahl im Speicherplatz 0040 mit 10 unter Verwendung von Links-Verschiebungen und Additionen[2]. Beachten Sie, daß ASL A den Inhalt des Akkumulators mit 2 multiplizieren. Dies gestattet Ihnen den Inhalt des Akkumulators mit kleinen Dezimalzahlen und wenigen Befehlen zu multiplizieren. Wie würden Sie dieses Verfahren verwenden, um mit 16 zu multiplizieren? Mit 12? Mit 7?

BCD-Zahlen benötigen etwa 20% mehr Speicher als Binärzahlen. Die Darstellung von 0 bis 999 benötigt 3 BCD-Stellen (12 Bits) und 10 Bits in binär (da $2^{10} = 1024 \approx 1000$).

Umwandlung einer Binärzahl in eine ASCII-Reihe

Zweck: Wandle die 8-Bit-Binärzahl im Speicherplatz 0041 in acht ASCII-Zeichen (entweder ASCII 0 oder ASCII 1) in den Speicherplätzen 0042 bis 0049(das höchstwertige Bit liegt in 0042) um.

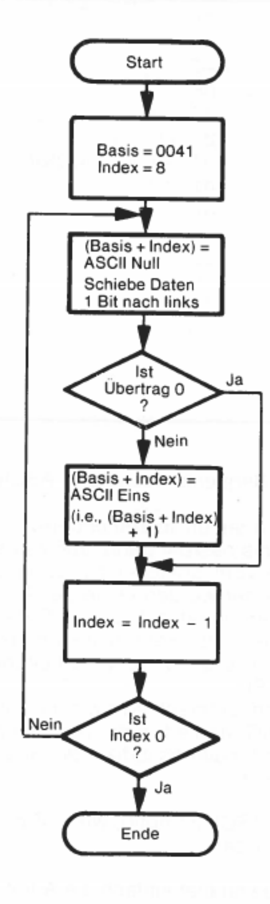
Beispiel:

(0041) = D2 = 11010010

Ergebnis:

(0042) = 31 ASCII 1
(0043) = 31 ASCII 1
(0044) = 30 ASCII 0
(0045) = 31 ASCII 1
(0046) = 30 ASCII 0
(0047) = 30 ASCII 0
(0048) = 31 ASCII 1
(0049) = 30 ASCII 0

Flußdiagramm:



Quellprogramm:

```
         LDA   $41       ;HOLE DATEN
         LDX   #8        ;ANZAHL DER BITS = 8
         LDY   #'0       ;HOLE ASCII NULL ZUR SPEICHERUNG IN
                         ;  REIHE
CONV     STY   $41,X     ;SPEICHERE ASCII NULL IN REIHE
         LSR   A         ;IST NÄCHSTES DATENBIT NULL
         BCC   COUNT
         INC   $41,X     ;NEIN, MACHE STRING-ELEMENT ASCII EINS
COUNT    DEX             ;ZÄHLE BITS
         BNE   CONV
         BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A5		LDA	$41
0001	41			
0002	A2		LDX	#8
0003	08			
0004	A0		LDY	#'0
0005	30			
0006	94	CONV	STY	$41,X
0007	41			
0008	4A		LSR	A
0009	90		BCC	COUNT
000A	02			
000B	F6		INC	$41,X
000C	41			
000D	CA	COUNT	DEX	
000E	D0		BNE	CONV
000F	F6			
0010	00		BRK	

Die ASCII-Ziffer bilden eine Sequenz, deshalb ist ASCII 1 = ASCII 0 + 1.

Der INC-Befehl kann zum direkten Inkrementieren eines Speicherplatzes verwendet werden. Die Ersparnis besteht darin, daß keine speziellen Befehle erforderlich sind, um die Daten vom Speicher zu laden oder das Ergebnis in den Speicher zurückzulegen. Ferner werden keine der Anwender-Register (A, X und Y) zerstört. Die CPU muß jedoch tatsächlich die Daten vom Speicher laden, sie in einem zeitweiligen Register aufbewahren, inkrementieren und das Ergebnis in den Speicher zurückladen. Die gesamte Verarbeitung der Daten geschieht in Wirklichkeit innerhalb der CPU.
Beachten Sie vor allem den Unterschied zwischen INX und einem Befehl wie INC $41,X. Der Befehl INC addiert Eins zum Inhalt des Indexregisters X. INC $41,X addiert Eins zum Inhalt des indizierten Speicherplatzes, er hat keinen Einfluß auf das Indexregister X.

Eine Umwandlung binär in ASCII ist nötig, wenn Zahlen in Binärform auf einem ASCII-Gerät ausgedruckt werden.

Die Umwandlung in ASCII beinhaltet einfach die Addition von ASCII Null (30_{16}).

AUFGABEN

1) ASCII in hexadezimal

Zweck: Wandle den Inhalt des Speicherplatzes 0040 in eine Hexadezimalzahl um und speichere das Ergebnis in den Speicherplatz 0041. Es werde angenommen, daß der Speicherplatz 0040 die ASCII-Darstellung einer Hexadezimalziffer enthält (7 Bits mit MSB 0).

Beispiele:

a.	(0040)	=	43	ASCII C
Ergebnis:	(0041)	=	0C	
b.	(0040)	=	36	ASCII 6
Ergebnis:	(0041)	=	06	

2) Sieben-Segment in dezimal

Zweck: Wandle den Inhalt des Speicherplatzes 0040 von einem Sieben-Segment-Code in eine Dezimalzahl im Speicherplatz 0041 um. Wenn der Speicherplatz 0040 keinen gültigen Sieben-Segment-Code enthält, setze Speicherplatz 0041 auf FF_{16}. Verwende die Sieben-Segment-Tabelle, die bei dem Beispiel über die Umwandlung dezimal in Sieben-Segment angegeben wurde und versuche die Codes in Übereinstimmung zu bringen.

Beispiele:

a.	(0040)	=	4F
Ergebnis:	(0041)	=	03
b.	(0040)	=	28
Ergebnis:	(0041)	=	FF

3) Dezimal in ASCII

Zweck: Wandle den Inhalt des Speicherplatzes 0040 von einer Dezimalziffer in ein ASCII-Zeichen um und speichere das Ergebnis in den Speicherplatz 0041. Wenn die Zahl im Speicherplatz 0040 keine Dezimalziffer ist, setze den Inhalt des Speicherplatzes 0041 auf ein ASCII-Leerzeichen (20_{16}).

Beispiele:

a.	(0040)	=	07	
Ergebnis:	(0041)	=	37	ASCII 7
b.	(0040)	=	55	
Ergebnis:	(0041)	=	20	ASCII SPACE

4) Binär in BCD

Zweck: Wandle den Inhalt des Speicherplatzes 0040 in zwei Dezimalstellen im Speicherplatz 0041 und 0042 um (höchstwertige Stelle in 0041). Die Zahl im Speicherplatz 0040 besitzt kein Vorzeichen und ist kleiner als 100.

Beispiele:

a.	(0040)	=	1D	(29 dezimal)
Ergebnis:	(0041)	=	02	
	(0042)	=	09	
b.	(0040)	=	47	(71 dezimal)
Ergebnis:	(0041)	=	07	
	(0042)	=	01	

5) Binärzahl in ASCII-Reihe

Zweck: Wandle die acht ASCII-Zeichen in den Speicherplätzen 0042 bis 0049 in eine 8-Bit-Binärzahl im Speicherplatz 0041 um (das höchstwertige Bit liegt in 0042). Lösche den Speicherplatz 0040, wenn alle ASCII-Zeichen entweder ASCII 1 oder ASCII 0 sind und setze anderenfalls auf FF_{16}.

Beispiele:

a.	(0042)	=	31	ASCII 1
	(0043)	=	31	ASCII 1
	(0044)	=	30	ASCII 0
	(0045)	=	31	ASCII 1
	(0046)	=	30	ASCII 0
	(0047)	=	30	ASCII 0
	(0048)	=	31	ASCII 1
	(0049)	=	30	ASCII 0
Ergebnis:	(0041)	=	D2	
	(0040)	=	00	

b. gleich wie "a" mit Ausnahme:

	(0045)	=	37	ASCII 7
Ergebnis:	(0040)	=	FF	

LITERATUR

1) Allison, D. R., "A Design Philosophie for Microcomputer Architectures". Computer, Februar 1977, Seite 35 – 41. Ein außerordentlich empfehlenswerter Artikel.
2) Weitere Verfahren für die Umwandlung BCD in binär werden besprochen in J.A. Tabb and M.L. Roginsky, "Microprocessor Algorithms Make BCD-Binary Conversions Super-fast," EDN, January 5, 1977, pp. 46-50 and in J.B. Peatman, Microcomputer-based Design, (New York: McGraw-Hill, 1977, pp. 400-406.

Kapitel 8
ARITHMETISCHE AUFGABEN

Die meisten arithmetischen Aufgaben bei Mikroprozessor-Anwendungen bestehen in der Verarbeitung von Binär- oder Dezimalzahlen, die aus mehreren Worten bestehen. Eine Dezimal-Korrektur (decimal adjust) oder einige andere Hilfsmittel zur Ausführung dezimaler Arithmetik ist häufig der einzige arithmetische Befehl, der neben der grundlegenden Addition und Subtraktion vorgesehen ist. Andere arithmetische Operationen muß man mit Befehlssequenzen ausführen.

Binär-Arithmetik mit mehrfacher Genauigkeit erfordert einfache Wiederholungen der grundlegenden Einwort-Befehle. Das Übertrags-Bit transferiert Informationen zwischen Worten. Addition mit Übertrag und Subtraktion mit Übertrag verwenden die Information von den vorhergehenden arithmetischen Operationen. Man muß sorgfältig darauf achten, den Übertrag zu löschen, bevor man die ersten Worte bearbeitet (offensichtlich gibt es keinen Übertrag in, oder "Borgen" von den niedrigstwertigen Bits).

Die Dezimal-Arithmetik ist eine so häufige Aufgabe für Mikroprozessoren, daß die meisten spezielle Befehle für diesen Zweck besitzen. Diese Befehle können entweder dezimale Operationen direkt ausführen oder die Ergebnisse von binären Operationen in die entsprechende Dezimalform anordnen. Die Dezimal-Arithmetik ist wesentlich in Anwendungen wie Registrierkassen, Rechner, Auftrags-Eingabesysteme und Bank-Terminals.

Man kann Multiplikation und Division als eine Serie von Additionen und Subtraktionen ausführen. Operationen an Doppelworten sind erforderlich, da eine Multiplikation ein Ergebnis liefert, das doppelt so lang wie die Operanden sind, während ähnlich eine Division die Länge des Ergebnisses verringert. Multiplikationen und Divisionen benötigen viel Zeit, wenn sie von der Software ausgeführt werden, da hierfür wiederholte arithmetische und Schiebe-Operationen erforderlich sind. Natürlich ist eine Multiplikation oder Division durch eine Potenz von 2 sehr einfach, da derartige Operationen mit einer entsprechenden Anzahl von arithmetischen Rechts- oder Links-Verschiebungen ausgeführt werden können.

BEISPIELE

Addition mit mehrfacher Genauigkeit

Zweck: Addiere zwei Binärzahlen aus mehreren Worten. Die Länge der Zahlen (in Bytes) befindet sich im Speicherplatz 0040, die Zahl selbst beginnt (höchstwertige Bits zuerst) entsprechend in den Speicherplätzen 0041 und 0051, und die Summe ersetzt die Zahl, die im Speicherplatz 0041 beginnt.

Beispiel:

(0040) = 04

(0041) = 2F
(0042) = 5B
(0043) = A7
(0044) = C3

(0051) = 14
(0052) = DF
(0053) = 35
(0054) = B8

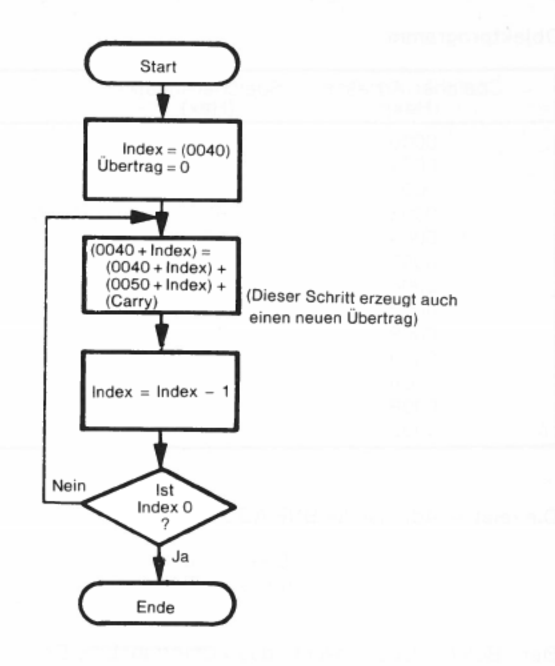
Ergebnis: (0041) = 44

(0042) = 3A
(0043) = DD
(0044) = 7B

das heißt:

```
 2F5BA7C3
+14DF35B8
 443ADD7B
```


Flußdiagramm:



Quellprogramm:

```
        LDA   $40     ;INDEX = LÄNGE DER REIHEN
        CLC           ;LÖSCHE ÜBERTRAG BEI START
ADDW    LDA   $40,X   ;HOLE BYTE VON REIHE 1
        ADC   $50,X   ;ADDIERE BYTE VON REIHE 2
        STA   $40,X   ;SPEICHERE ERGEBNIS IN REIHE 1
        DEX
        BNE   ADDW    ;SETZE FORT BIS ALLE BYTES ADDIERT
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
0000	A6		LDX	$40
0001	40			
0002	18		CLC	
0003	B5	ADDW	LDA	$40,X
0004	40			
0005	75		ADC	$50,X
0006	50			
0007	95		STA	$40,X
0008	40			
0009	CA		DEX	
000A	D0		BNE	ADDW
000B	F7			
000C	00		BRK	

Die relative Adresse für BNE ADDW ist

$$\begin{array}{r} 0003 \\ -000C \end{array} = \begin{array}{r} 03 \\ +F4 \\ \hline F7 \end{array}$$

Der Befehl CLC löscht das Übertragsbit. Der Übertrag muß gelöscht werden, da bei der Addition der niedrigstwertigen Bytes kein Übertrag enthalten sein kann.

Der Befehl ADC, (ADD WITH CARRY) enthält den Übertrag von den vorhergehenden Worten der Addition. ADC ist der einzige Befehl in der Schleife, der den Übertrag beeinflußt. Beachten Sie, daß weder Inkrementiert- noch Dekrementier-Befehle (DEC, DEX, DEY, INC, INX, INY) den Übertrag nicht beeinflußen.

DEZIMALE GENAUIGKEIT IN BINÄR

Dieses Programm verwendet den gleichen Index mit zwei verschiedenen Basis-Adressen, um die beiden Reihen zu handhaben. Die Reihen können überall in den Speicher gelegt werden. Ferner wäre es nicht schwierig, das Ergebnis in eine dritte Reihe abzuspeichern.

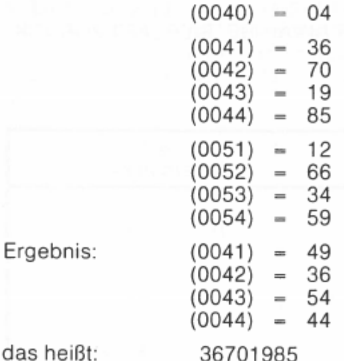
Dieses Verfahren kann Binärzahlen jeder Länge addieren. Beachten Sie, daß zehn binäre Bits drei dezimalen Stellen entsprechen, da $2^{10} = 1024 \approx 1000$. Daher kann man die Anzahl der erforderlichen Bits berechnen, um eine bestimmte Genauigkeit in Dezimalstellen anzugeben. Zum Beispiel benötigt eine zwölfstellige Dezimalzahl

$$12 \times \frac{10}{3} = 40 \text{ Bits}$$

Dezimale Addition

Zweck: Addiere zwei Dezimal(BCD)-Zahlen aus mehreren Worten. Die Länge der Zahlen befindet sich im Speicherplatz 0040, die Zahlen selbst beginnen (höchstwertige Bits zuerst) entsprechend in den Speicherplätzen 0041 und 0051, und die Summe ersetzt die Zahl, die im Speicherplatz 0041 beginnt.

Beispiel:



(0040) = 04

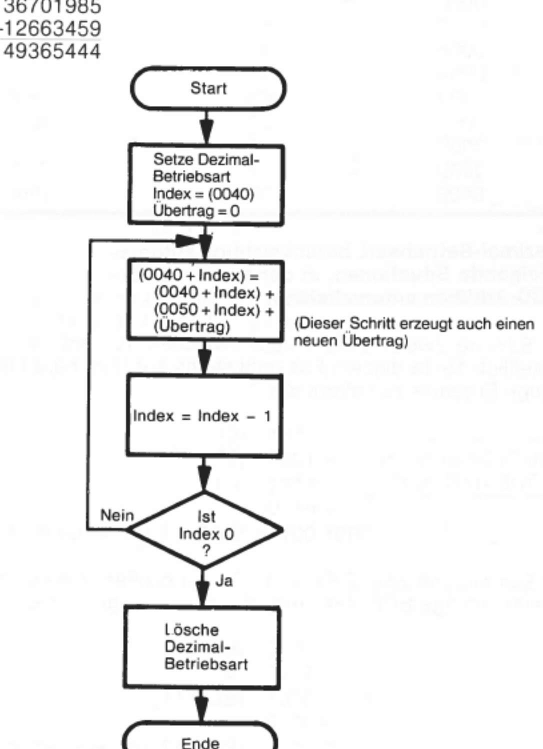
(0041) = 36
(0042) = 70
(0043) = 19
(0044) = 85

(0051) = 12
(0052) = 66
(0053) = 34
(0054) = 59

Ergebnis:
(0041) = 49
(0042) = 36
(0043) = 54
(0044) = 44

das heißt:
36701985
+12663459
49365444

Flußdiagramm:



Quellprogramm:

```
        SED              ;MACHE DIE ARITHMETIK DEZIMAL
        LDX   $40        ;INDEX = LÄNGE DER REIHEN
        CLC              ;LÖSCHE ÜBERTRAG BEI START
ADDW    LDA   $40,X      ;HOLE ZWEI STELLEN VON REIHE 1
        ADC   $50,X      ;ADDIERE ZWEI STELLEN VON REIHE 2
        STA   $40,X      ;SPEICHERE ERGEBNIS IN REIHE 1
        DEX
        BNE   ADDW       ;SETZE FORT BIS ALLE STELLEN ADDIERT
        CLD              ;KEHRE ZUR BINÄR-BETRIEBSART ZURÜCK
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
0000	F8		SED	
0001	A6		LDX	$40
0002	40			
0003	18		CLC	
0004	B5	ADDW	LDA	$40,X
0005	40			
0006	75		ADC	$50,X
0007	50			
0008	95		STA	$40,X
0009	40			
000A	CA		DEX	
000B	D0		BNE	ADDW
000C	F7			
000D	D8		CLD	
000E	00		BRK	

DEZIMAL-BETRIEBSART DES 6502

Die Dezimal-Betriebsart berücksichtigt automatisch folgende Situationen, in denen sich Binär- und BCD-Addition unterscheiden.

1) Die Summe zweier Ziffern liegt zwischen 10 und einschließlich 15. In diesem Fall muß sechs zur Summe addiert werden, um das richtige Ergebnis zu liefern, d.h.:

```
     0101   (5)
   + 1000   (8)
     1101   (D)
   + 0110
0001 0011   (BCD 13, das korrekt ist)
```

2) Die Summe von zwei Ziffern ist 16 oder größer. In diesem Fall ist das Ergebnis eine richtige BCD-Zahl, jedoch sechs weniger, als es sein sollte, d.h.

```
     1000   (8)
   + 1001   (9)
0001 0001   (BCD 11)
   + 0110
0001 0111   (BCD 17, das korrekt ist)
```


Sechs muß in beiden Fällen addiert werden. Jedoch kann Fall 1 durch die Tatsache erkannt werden, daß die Summe keine BCD-Ziffer ist. Sie liegt zwischen 10 und 15 (oder A und F hexadezimal). Fall 2 kann nur durch die Tatsache erkannt werden, daß der Übertrag aus der Addition einer Stelle eins ist, da das Ergebnis eine gültige BCD-Zahl ist.

Wenn das Flag für die Dezimal-Betriebsart gesetzt ist, wird die gesamte Arithmetik in dezimaler Form ausgeführt. Dies beinhaltet sowohl Subtraktionen als auch Additionen, unabhängig davon, welche Adressier-Art angewendet wird.

GRENZEN DER DEZIMAL-BETRIEBSART

Die Inkremtier- und Dekrementier-Befehle erzeugen jedoch binäre Ergebnisse, auch wenn das Flag für die dezimale Betriebsart gesetzt ist.
Daher kann DEC, DEX, DEY, INC, INY und INX nur für binäre Zähler verwendet werden. Um zum Beispiel einen Dezimalzähler im Speicherplatz 0040 zu inkrementieren, müssen Sie folgende Sequenz verwenden.

```
SED          ;MACHE ARITHMETIK DEZIMAL
LDA   $40    ;HOLE ZÄHLER
CLC          ;HINDERE ÜBERTRAG AM BEEINFLUSSEN
             ;  DER ADDITION
ADC   #1     ;INKREMENTIERE ZÄHLER (DEZIMAL)
STA   $40
CLD          ;KEHRE ZU BINÄRER BETRIEBSART
             ;  ZURÜCK
```

Die Befehle SED, CLC und CLD können nicht erforderlich sein, wenn andere Teile des Programmes die Statusflags entsprechend setzen.

Subtraktionen in der Dezimal-Betriebsart erzeugen korrekte BCD-Ergebnisse mit dem Übertrag, der ein invertiertes "Borgen" darstellt. Wenn zum Beispiel der Akkumulator 03 enthält, der adressierte Speicherplatz 27 und der Übertrag 1 ist, so wird nach Ausführung eines SBC-Befehls der Akkumulator 76 enthalten und der Übertrag 0 sein. Wie bei der binären Betriebsart bedeutet ein Übertrag von 0, daß ein Borgen erzeugt wurde.

Das Vorzeichen-Bit hat keine Bedeutung nach Additionen und Subtraktionen, wenn das Flag für die Dezimal-Betriebsaart gesetzt ist. Es gibt das Ergebnis einer binären Operation wieder, nicht das einer dezimalen Operation. In der kürzlich erwähnten Situation (03-27) wird das Vorzeichen-Bit gesetzt (wie es der Fall sein würde, wenn die Zahlen binär wären), auch wenn das dezimale Ergebnis (76) ein höchst-wertiges Bit von 0 besitzt.

GENAUIGKEIT IN BINÄR UND BCD

Dieses Verfahren kann Dezimal- (BCD-)Zahlen jeder Länge verarbeiten. Hier sind vier binäre Bits für jede Dezimalstelle erforderlich, so daß 12 x 4 = 48 Bits eine zwölfstellige Genauigkeit erfordert, im Gegensatz zu 40 Bits im binären Fall. Dies sind sechs 8-Bit-Worte anstatt fünf.

8-Bit-Binärmultiplikation

Zweck: Multipliziere die 8-Bit-Zahl ohne Vorzeichen im Speicherplatz 0040 mit der 8-Bit-Zahl ohne Vorzeichen im Speicherplatz 0041. Plaziere die acht niedrigstwertigen Bits des Ergebnisses in den Speicherplatz 0042 und die acht höchstwertigen Bits in den Speicherplatz 0043.

Beispiele:

a. (0040) = 03
(0041) = 05

Ergebnis: (0042) = 0F
(0043) = 00
oder in dezimal 3 x 5 = 15

(0040) = 6F
(0041) = 61

Ergebnis: (0042) = 0F
(0043) = 2A
oder 111 x 97 = 10 767

Man kann eine Multiplikation auf einem Computer auf die gleiche Weise ausführen, wie man es manuell bei einer langen Multiplikation macht. Da es sich um Binärzahlen handelt, besteht die einzige Aufgabe darin, mit 0 oder mit 1 zu multiplizieren. Die Multiplikation mit null ergibt offensichtlich null als Ergebnis, während eine Multiplikation mit eins die gleiche Zahl ergibt, mit der man begonnen hat (dem Multiplikand). Daher kann jeder Schritt in einer Binär-Multiplikation auf die folgende Operation reduziert werden:

MULTIPLIKATIONS-ALGORITHMUS

Wenn das momentane Bit im Multiplikator 1 ist, addiere den Multiplikanden zum Teilprodukt.

Das einzige verbleibende Problem besteht darin, daß man alles zu jeder Zeit korrekt gruppiert. Die folgenden Operationen führen diese Aufgabe aus:

1) Verschiebe Multiplikator um ein Bit nach links, so daß das zu prüfende Bit in den Übertrag plaziert wird.
2) Verschiebe Produkt um ein Bit nach links, so daß die nächste Addition korrekt angeordnet ist.

Der vollständige Vorgang für eine Binär-Multiplikation ist wie folgt:

Schritt 1 – Initialisierung
Produkt = 0
Zähler = 8

Schritt 2 – Schiebe Produkt so, daß es richtig angeordnet ist.
Produkt = 2 x Produkt (LSB = 0)

Schritt 3 – Schiebe Multiplikator so, daß das Bit in den Übertrag gelangt.
Multiplikator = 2 x Multiplikator

Schritt 4 – Addiere Multiplikand zum Produkt, wenn der Übertrag 1 ist.
Wenn Übertrag = 1, Produkt = Produkt + Multiplikand

Schritt 5 – Dekrementiere Zähler und prüfe auf null
Zähler = Zähler –1
Wenn Zähler ≠ 0, gehe zu Schritt 2

Im Falle des Beispiels b, in dem der Multiplikator 61_{16} und der Multiplikand $6F_{16}$ ist, arbeitet das Verfahren wie folgt:

Initialisierung:

Produkt	0000
Multiplikator	61
Multiplikand	6F
Zähler	08

Nach der ersten Wiederholung der Schritte 2 – 5:

Produkt	0000
Multiplikator	C2
Multiplikand	6F
Zähler	07
Übertrag vom Multiplikator	0

Nach der zweiten Wiederholung:

Produkt	006F
Multiplikator	84
Multiplikand	6F
Zähler	06
Übertrag vom Multiplikator	1

Nach der dritten Wiederholung:

Produkt	014D
Multiplikator	08
Multiplikand	6F
Zähler	05
Übertrag vom Multiplikator	1

Nach der vierten Wiederholung:

Produkt	029A
Multiplikator	10
Multiplikand	6F
Zähler	04
Übertrag vom Multiplikator	0

Nach der fünften Wiederholung:

Produkt	0534
Multiplikator	20
Multiplikand	6F
Zähler	03
Übertrag vom Multiplikator	0

Nach der sechsten Wiederholung:

Produkt	0A68
Multiplikator	40
Multiplikand	6F
Zähler	02
Übertrag vom Multiplikator	0

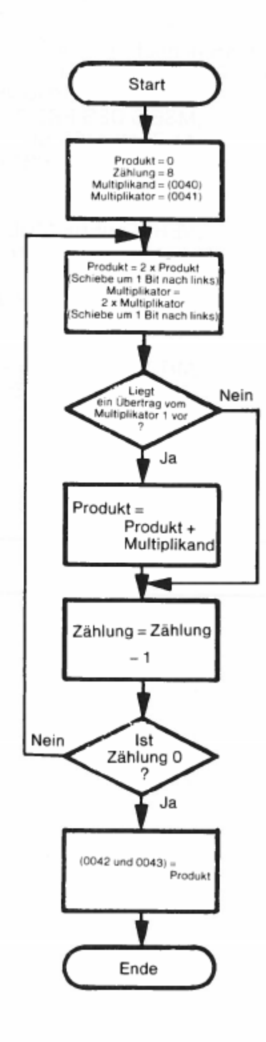
Nach der siebten Wiederholung:

Produkt	14D0
Multiplikator	80
Multiplikand	6F
Zähler	01
Übertrag vom Multiplikator	0

Nach der achten Wiederholung:

Produkt	2A0F
Multiplikator	00
Multiplikand	6F
Zähler	00
Übertrag vom Multiplikator	1

Flußdiagramm:



Quellprogramm:

```
        LDA   #0        ;LSB'S DES PRODUKTS = NULL
        STA   $43       ;MSB'S DES PRODUKTS = NULL
        LDX   #8        ;ANZAHL DER BITS IM MULTIPLIKATOR = 8
SHIFT   ASL   A         ;VERSCHIEBE PRODUKT UM EIN BIT NACH
                        ;  LINKS
        ROL   $43
        ASL   $41       ;VERSCHIEBE MULTIPLIKATOR NACH LINKS
        BCC   CHCNT     ;KEINE ADDITION WENN NÄCHSTES BIT
                        ;  NULL
        CLC             ;ADDIERE MULTIPLIKANT ZUM PRODUKT
        ADC   $40
        BCC   CHCNT
        INC   $43       ;MIT ÜBERTRAG FALLS ERFORDERLICH
CHCNT   DEX             ;SCHLEIFE, BIS 8 BITS MULTIPLIZIERT SIND
        BNE   SHIFT
        STA   $42       ;SPEICHERE LSB'S DES PRODUKTS
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A9		LDA	#0
0001	00			
0002	85		STA	$43
0003	43			
0004	A2		LDX	#8
0005	08			
0006	0A	SHIFT	ASL	A
0007	26		ROL	$43
0008	43			
0009	06		ASL	$41
000A	41			
000B	90		BCC	CHCNT
000C	07			
000D	18		CLC	
000E	65		ADC	$40
000F	40			
0010	90		BCC	CHCNT
0011	02			
0012	E6		INC	$43
0013	43			
0014	CA	CHCNT	DEX	
0015	D0		BNE	SHIFT
0016	EF			
0017	85		STA	$42
0018	42			
0019	00		BRK	

Neben seiner offensichtlichen Verwendung in Rechnern und Registrierkassen ist die Multiplikation ein wesentlicher Teil aller Signalverarbeitungs- und Steuer-Algorithmen.

Die Geschwindigkeit, mit der Multiplikationen ausgeführt werden können, bestimmen die Nützlichkeit einer CPU in der Prozess-Steuerung, Signal-Erkennung und Signal-Analyse.

Dieser Algorithmus benötigt zwischen 170 und 280 Taktzyklen, um auf einem 6502 zu multiplizieren. Die genaue Zeit hängt von der Anzahl der 1-Bits im Multiplikator ab. Andere Algorithmen können die durchschnittliche Zeit etwas verringern, jedoch werden 250 Taktzyklen für eine Software-Multiplikation typisch sein.

Einige Mikroprozessoren (wie der 9900, 8086 und Z8000) besitzen Befehle für eine Hardware-Multiplikation, die etwas schneller ist, wobei jedoch die maximale Geschwindigkeit zusätzliche externe Hardware benötigt.

8-Bit-Binär-Division

Zweck: Dividiere die 16-Bit-Zahl ohne Vorzeichen in den Speicherplätzen 0040 und 0041 (höchstwertige Bits in 0041) durch die 8-Bit-Zahl ohne Vorzeichen im Speicherplatz 0042. Die Zahlen sind normiert, so daß 1 die höchstwertigen Bits sowohl des Dividenden und des Divisors null sind und 2 die Zahl im Speicherplatz 0042 größer als die Zahl im Speicherplatz 0041 ist. Daher ist der Quotient eine 8-Bit-Zahl. Speichere den Quotienten in den Speicherplatz 0043 und den Rest in den Speicherplatz 0044.

Beispiele:

a. (0040) = 40 (64 dezimal)
(0041) = 00
(0042) = 08

Ergebnis: (0043) = 08
(0044) = 00
i.e. 64/8 = 8

b. (0040) = 6D (12 909 dezimal)
(0041) = 32
(0042) = 47 (71 dezimal)

Ergebnis: (0043) = B5 (181 dezimal)
(0044) = 3A (58 dezimal)
i.e. 12 909/71 = 181 mit einem Rest von 58

DIVISIONS-ALGORITHMUS

Man kann eine Division mit einem Computer genauso ausführen, wie man es mit Bleistift und Papier machen würde, d.h., durch die versuchsweise Verwendung von Subtraktionen.
Da es sich um Binärzahlen handelt, besteht die einzige Frage darin, ob das Bit im Quotienten 0 oder 1 ist, d.h., ob der Divisor von dem subtrahiert werden kann, was vom Dividenden übrig gelassen wurde. Jeder Schritt in einer Binär-Division kann auf folgende Operationen reduziert werden:

> Wenn der Divisor von den acht höchstwertigen Bits des Dividenden ohne "Borgen" subtrahiert werden kann, ist das entsprechende Bit im Quotien-ten 1. Andernfalls ist es 0.

Das einzige verbleibende Problem liegt im richtigen Anordnen des Dividenden und Quotienten. Man kann dies vor jeder versuchsweisen Subtraktion, durch logische Linksverschiebung (um ein Bit) des Dividenden und Quotienten ausführen. Dividend und Quotient können sich ein 16-Bit-Register teilen, da das Verfahren ein Bit des Dividenden zur gleichen Zeit löscht, in der es ein Bit des Quotienten bestimmt.

Der vollständige Vorgang für binäre Divisionen ist:

Schritt 1 – Initialisierung
Quotient = 0
Zähler = 8

Schritt 2 – Verschiebe Dividenden und Quotienten sod, daß sie entsprechend angeordnet sind.
Dividend = 2 x Dividend
Quotient = 2 x Quotient

Schritt 3 – Führe Subtraktion versuchsweise aus. Wenn kein "Borgen", addiere 1 zum Quotienten. Wenn 8 MSBs des Dividenden ⩾ Divisor, dann sind die MSBs des Dividenden = MBS des Dividenden – Divisor, Quotient = Quotient + 1.

Schritt 4 – Dekrementiere Zähler und prüfe auf null
Zähler = Zähler –1
wenn der Zähler ≠ 0, gehe zu Schritt 2
Rest = MBSs des Dividenden

Im Falle des Beispiels b, in dem der Dividend $326D_{16}$ und der Divisor 47_{16} ist, arbeitet das Verfahren wie folgt:

Initialisierung:

Dividend	326D
Divisor	47
Quotient	00
Zähler	00

Nach der ersten Wiederholung der Schritte 2 – 4: (Beachten Sie, daß der Dividend vor der versuchsweisen Subtraktion verschoben wird).

Dividend	1DDA
Divisor	47
Quotient	01
Zähler	07

Nach der zweiten Wiederholung der Schritte 2 – 4:

Dividend	3BB4
Divisor	47
Quotient	02
Zähler	06

Nach der dritten Wiederholung:

Dividend	3068
Divisor	47
Quotient	05
Zähler	05

Nach der vierten Wiederholung:

Dividend	19D0
Divisor	47
Quotient	0B
Zähler	04

Nach der fünften Wiederholung:

Dividend	33A0
Divisor	47
Quotient	16
Zähler	03

Nach der sechsten Wiederholung:

Dividend	2040
Divisor	47
Quotient	2D
Zähler	02

Nach der siebten Wiederholung:

Dividend	4080
Divisor	47
Quotient	5A
Zähler	01

Nach der achten Wiederholung:

Dividend	3A00
Divisor	47
Quotient	B5
Zähler	00

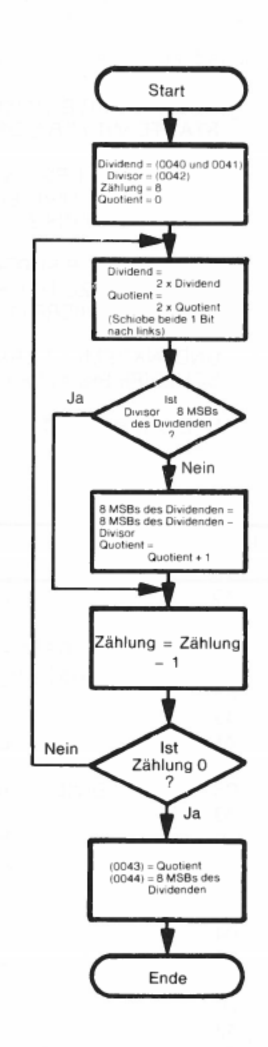
Daher ist der Quotient B5 und der Rest 3A.

Die MSBs des Dividenden und Divisors sind angenommen null, um die Berechnungen zu vereinfachen (die Verschiebung vor der versuchsweisen Subtraktion würde andernfalls das MSB des Dividenden in den Übertrag plazieren). Aufgaben, die nicht in dieser Form vorliegen, müssen vereinfacht werden, indem Teile des Quotienten entfernt werden, die einen Überlauf eines 8-Bit-Wortes bewirken würden. Zum Beispiel:

$$\frac{1024}{3} = \frac{400\ (\text{Hex})}{3} = 100 + \frac{100\ (\text{Hex})}{3}$$

Die letzte Aufgabe befindet sich nun in der richtigen Form. Eine zusätzliche Division kann erforderlich sein.

Flußdiagramm:



Quellprogramm :

```
        LDX   #8        ;ANZAHL DER BITS IM DIVISOR = 8
        LDA   $40       ;STARTE MIT LSB'S DES DIVIDENDEN
        STA   $43
        LDA   $41       ;HOLE MSB'S DES DIVIDENDEN
DIVID   ASL   $43       ;VERSCHIEBE DIVIDEND, QUOTIENT UM 1
                        ;  BIT NACH LINKS
        ROL   A
        CMP   $42       ;KANN DIVISOR SUBTRAHIERT WERDEN?
        BCC   CHCNT     ;NEIN, GEHE ZU NÄCHSTEM SCHRITT
        SBC   $42       ;JA, SUBTRAHIERE QUOTIENTEN (ÜBER-
                        ;  TRAG = 1)
        INC   $43       ;UND INKREMENTIERE QUOTIENT UM 1
CHCNT   DEX             ;SCHLEIFE BIS ALLE 8 BITS VERARBEITET
        BNE   DIVID
        STA   $44       ;SPEICHERE REST
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A2		LDX	#8
0001	08			
0002	A5		LDA	$40
0003	40			
0004	85		STA	$43
0005	43			
0006	A5		LDA	$41
0007	41			
0008	06	DIVID	ASL	$43
0009	43			
000A	2A		ROL	A
000B	C5		CMP	$42
000C	42			
000D	90		BCC	CHCNT
000E	04			
000F	E5		SBC	$42
0010	42			
0011	E6		INC	$43
0012	43			
0013	CA	CHCNT	DEX	
0014	D0		BNE	DIVID
0015	F2			
0016	85		STA	$44
0017	44			
0018	00		BRK	

Die Division wird in Rechnern, Terminals, bei der Prüfung von Übertragungsfehlern, Steuer-Algorithmen und zahlreichen anderen Anwendungen eingesetzt.

Der Algorithmus benötigt zwischen 150 und 230 Mikrosekunden, um eine Division auf einem 6502 mit einer Taktfrequenz von 1MHz auszuführen. Die genaue Zeit hängt von der Anzahl der 1-Bits im Quotienten ab. Andere Algorithmen können die durchschnittliche Zeit etwas verringern, jedoch 200 Mikrosekunden sind typisch für eine Software-Division.

Die Befehle ASL $43 und ROL A ergeben zusammen eine arithmetische Linksverschiebung mit 16 Bits des Dividenden (MSBs in A). Der ROL-Befehl nimmt das Bit auf, das der ASL-Befehl im Übertrag zurückläßt.

Eine 8-Bit-Subtraktion ist erforderlich, da es keinen einfachen Weg für eine 16-Bit-Subtraktion oder Vergleich gibt.

Der Speicherplatz 0043 und der Akkumulator beinhalten sowohl den Dividenden als auch den Quotienten. Der Quotient ersetzt einfach den Dividenden im Speicherplatz 0043, während der Dividend arithmetisch nach links verschoben wird.

Wir brauchen uns nicht um den Übertrag in dem SBC-Befehl zu kümmern. Er muß 1 sein, da andernfalls BCC eine Verzweigung bewirkt haben würde. Erinnern Sie sich daran, daß ein Übertragswert von 1 keinen Einfluß auf das Ergebnis eines SBC-Befehls hat, da der Übertrag ein invertiertes "Borgen" darstellt.

Selbst-prüfende Zahlen ("Double Add Double Mod 10")

Zweck: Berechne eine Prüfsummen-Ziffer von einer Reihe von BCD-Ziffern. Die Länge der Reihe der Ziffern (Anzahl der Worte) befindet sich im Speicherplatz 0041. Die Ziffernreihe (2 BCD-Ziffern in einem Wort) beginnt im Speicherplatz 0042. Berechne die Prüfsummen-Ziffer mit dem sogenannten "Double Add Double Mod 10"-Verfahren[1] und speichere diese in den Speicherplatz 0040.

SELBST-PRÜFENDE ZAHLEN

Das Verfahren "Double Add Double Mod 10" arbeitet wie folgt:

1) Lösche zu Beginn die Prüfsumme.
2) Multipliziere die führende Stelle mit 2 und addiere das Ergebnis zur Prüfsumme.
3) Addiere die nächste Stelle zur Prüsumme.
4) Setze den abwechselnden Vorgang fort, bis alle Stellen verwendet wurden.
5) Die niedrigstwertige Stelle der Prüfsumme ist die selbstprüfende Ziffer.

Selbstprüfende Ziffern werden gewöhnlich zur Identifizierung von Zahlen auf Kreditkarten, Inventur-Etiketten, Gepäck, Paketen etc. hinzugefügt, wenn sie von Computersystemen verarbeitet werden. Sie können auch in Routine-Mitteilungen, Identifizierungs-Dateien und anderen Anwendungen eingesetzt werden. Der Zweck der Ziffern besteht in einer Minimisierung von Eingabe-Fehlern, wie etwa vertauschte Stellen (69 anstatt 96), verschobene Ziffern (7260 anstatt 3726), Abweichungen von Ziffern um 1 (65 anstatt 64), etc. Man kann die selbstprüfenden Zahlen automatisch auf Richtigkeit der Eingabe überprüfen und man kann zahlreiche Fehler sofort eliminieren.

Die Analysis der selbstprüfenden Methode ist ziemlich komplex. Beispielsweise wird eine glatte Prüfsumme Vertauschungs-Fehler nicht finden (4 + 9 = 9 + 4). Der Algorithmus für das "Double Add Double Mod 10" wird einfache Vertauschungsfehler feststellen (2 x 4 + 9 = 17 ≠ 2 x 9 + 4), wird jedoch einige Fehler nicht aufdecken, wie etwa Vertauschungen einer geraden Anzahl von Stellen (367 anstatt 763). Trotzdem wird dieses Verfahren zahlreiche allgemeine Fehler finden! Der Wert dieser Methode hängt davon ab, welche Fehler sie feststellen wird und von der Wahrscheinlichkeit, mit der spezielle Fehler in einer Anwendung auftreten.

Beispielsweise wird das Ergebnis bei einer Reihe von Ziffern

549321

folgendes sein:

Prüfsumme = 5 x 2 + 4 + 9 x 2 + 3 + 2 x 2 + 1 = 40
Selbstprüfende Ziffer = 0 (niedrigstwertige Stelle der Prüfsumme)

Beachten Sie, daß eine fehlerhafte Eingabe wie 543921 eine unterschiedliche selbstprüfende Ziffer (4) erzeugt, jedoch fehlerhafte Eingaben wie 049321 oder 945321 nicht festgestellt würden.

Beispiele:

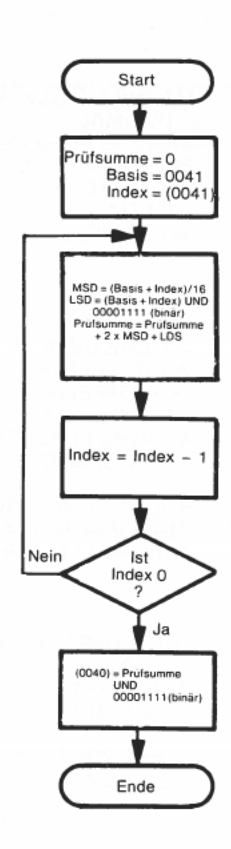
a.
(0041) = 03
(0042) = 36
(0043) = 68
(0044) = 51

Ergebnis:
Prüfsumme = 3 x 2 + 6 + 6 x 2 + 8 + 5 x 2 + 1 = 43
(0040) = 03

b.
(0041) = 04
(0042) = 50
(0043) = 29
(0044) = 16
(0045) = 83

Ergebnis:
Prüfsumme = 5 x 2 + 0 + 2 x 2 + 9 + 1 x 2 + 6 + 8 x 2 + 3 = 50
(0040) = 00

Flußdiagramm:



Quellprogramm:

```
        SED                 ;MACHE DIE GESAMTE ARITHMETIK
                            ;  DEZIMAL
        LDX   $41           ;INDEX = LÄNGE DER REIHE
        LDY   #0            ;PRÜFSUMME = NULL
CHKDG   LDA   $41,X         ;HOLE NÄCHSTE ZWEI STELLEN DER
                            ;  DATEN
        LSR   A             ;VERSCHIEBE NIEDRIGSTWERTIGE STELLE
        LSR   A
        LSR   A
        LSR   A
        STA   $40
        CLC                 ;LÖSCHE ÜBERTRAG VON VERSCHIEBUNG
        ADC   $40           ;VERDOPPLE HÖCHSTWERTIGE STELLE
        STY   $40           ;VERDOPPELN EINER ZIFFER ERZEUGT
                            ;  NIEMALS EINEN ÜBERTRAG
        ADC   $40           ;ADDIERE VERDOPPELTES MSD ZUR
                            ;  PRÜFSUMME
        STA   $40
        LDA   $41,X         ;HOLE NIEDRIGSTWERTIGE STELLE
        AND   #%00001111    ;(MASKIERE MSD AUS)
        CLC                 ;ADDIERE LSD ZUR PRÜFSUMME
        ADC   $40
        TAY
        DEX
        BNE   CHKDG         ;SETZE FORT, BIS ALLE ZIFFERN SUMMIERT
                            ;  SIND
        AND   #%00001111    ;BEWAHRE LSD DER SELBSTPRÜFENDEN
                            ;  ZAHL AUF
        STA   $40
        CLD                 ;KEHRE ZUR BINÄREN BETRIEBSART
                            ;  ZURÜCK
        BRK
```


Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	F8		SED	
0001	A6		LDX	$41
0002	41			
0003	A0		LDY	#0
0004	00			
0005	B5	CHKDG	LDA	$41,X
0006	41			
0007	4A		LSR	A
0008	4A		LSR	A
0009	4A		LSR	A
000A	4A		LSR	A
000B	85		STA	$40
000C	40			
000D	18		CLC	
000E	65		ADC	$40
000F	40			
0010	84		STY	$40
0011	40			
0012	65		ADC	$40
0013	40			
0014	85		STA	$40
0015	40			
0016	B5		LDA	$41,X
0017	41			
0018	29		AND	#%00001111
0019	0F			
001A	18		CLC	
001B	65		ADC	$40
001C	40			
001D	A8		TAY	
001E	CA		DEX	
001F	D0		BNE	CHKDG
0020	E4			
0021	29		AND	#%00001111
0022	0F			
0023	85		STA	$40
0024	40			
0025	D8		CLD	
0026	00		BRK	

Die Ziffern werden durch Verschieben und Maskieren entfernt. Vier logische Rechts-Verschiebungen sind erforderlich, um die höchstwertige Stelle zu separieren.

Die gesamte Arithmetik wird in dezimal ausgeführt. Erinnern Sie sich jedoch daran, daß DEX weiterhin ein binäres Ergebnis liefert.

Es gibt keine Probleme mit dem Übertrag vom Verdoppeln von Dezimalziffern, da das Ergebnis niemals größer als 18 sein kann. Sie können imstande sein, den letzten CLC-Befehl zu eliminieren, wenn Sie wissen, daß die zu summierenden Zahlen zu klein sind, um jeweils einen Übertrag zu erzeugen.

VERDOPPELN UND HALBIEREN VON DEZIMALZAHLEN

Sie können eine Dezimalzahl im Akkumulator verdoppeln, indem sie diese zu sich selbst in der dezimalen Betriebsart addieren.

Eine typische Sequenz wäre folgende (Verwendung des Speicherplatzes 0040 für Zwischenspeicherung):

```
        SED              ;MACHE ARITHMETIK DEZIMAL
        STA    $40
        CLC              ;VERHINDERE, DASS ÜBERTRAG ADDITION
                         ;  BEEINFLUSST
        ADC    $40       ;VERDOPPLE ZAHL
        CLD              ;KEHRE ZUR BINÄREN BETRIEBSART
                         ;  ZURÜCK
```

Sie können vielleicht die Befehle SED, CLC und CLD nicht benötigen, wenn andere Teile des Programmes die Flags für den Übertrag und die Dezimal-Betriebsart entsprechend setzen. Beachten Sie, daß Sie nicht ASL A zum Verdoppeln der Dezimalzahl verwenden können, da dieser Befehl ein binäres Ergebnis liefert, auch wenn das Flag für die Dezimal-Betriebsart gesetzt ist.

Man kann eine Dezimalzahl durch 2 dividieren, indem man sie einfach nach rechts logisch verschiebt und dann 3 von jeder Stelle subtrahiert, die 8 oder größer ist (da 10 BCD gleich ist 16 binär). Das folgende Programm dividiert eine Dezimalzahl im Speicherplatz 0040 durch 2 und plaziert das Ergebnis in den Speicherplatz 0041:

```
        LDA    $40          ;HOLE DEZIMALZAHL
        LSR    A            ;DIVIDIERE DURCH 2 IN BINÄR
        TAX
        AND    #%00001111   ;IST DIE NIEDRIGSTWERTIGE STELLE 8
                            ;  ODER GRÖSSER?
        CMP    #8
        BCC    DONE
        TAX
        SBC    #3           ;JA, SUBTRAHIERE 3 FÜR EINE DEZIMALE
                            ;  KORREKTUR
        TAX
DONE    STX    $41          ;SPEICHERE DURCH 2 DIVIDIERTE ZAHL
        BRK
```

Es gibt kein Problem mit dem Übertrag im SBC-Befehl, da dieser Befehl nur ausgeführt wird, wenn der Übertrag gesetzt ist. Erinnern Sie sich daran, daß SBC vom komplementierten Übertrag (1 – C) subtrahiert, so daß ein Übertrag von 1 niemals das Ergebnis beeinflußt.

Versuchen Sie dieses Programm und das Verfahren manuell an den Dezimalzahlen 28, 30 und 37. Haben Sie verstanden wie es arbeitet?

BINÄRE AUFRUNDUNG

Aufrunden ist sehr einfach, egal ob die Zahlen in binär oder dezimal vorliegen. Eine Binärzahl kann wie folgt aufgerundet werden:

> Wenn das höchstwertige Bit, das zu unterdrücken ist, gleich 1 ist, so addiere 1 zu den verbleibenden Bits. Andernfalls lasse die verbleibenden Bits unverändert.

Diese Regel arbeitet, da 1 in der Mitte zwischen 0 und 10 in binär liegt, so wie 5 in der Mitte bei Dezimalzahlen (Beachten Sie, daß 0.5 dezimal = 0.1 binär).

Daher wird das folgende Programm eine 16-Bit-Zahl in den Speicherplätzen 0040 und 0041 (MSBs in 0041) auf eine 8-Bit-Zahl im Speicherplatz 0040 aufrunden:

```
        LDA    $40       ;IST DAS MSB DES EXTRABYTE 1
        BPL    DONE
        INC    $41       ;JA, RUNDE MSB'S AUF
DONE    BRK
```

Wenn die Zahl größer ist als 16 Bits, muß die Aufrundung durch die anderen Bytes hindurch wenn nötig erfolgen.

Beachten Sie, daß wir BIT $40 anstatt LDA $40 verwenden könnten, da der BIT-Befehl das Vorzeichen-Flag entsprechend dem höchstwertigen Bit des adressierten Speicherplatzes setzt. Diese Lösung läßt den Akkumulator so wie er war, obwohl er die Statusflags ändert.

DEZIMALE AUFRUNDUNG

Dezimale Aufrundung ist etwas schwierig, da der Übergangspunkt nun BCD 50 ist und die Aufrundung ein dezimales Ergebnis liefern muß.
Die Regel lautet:

> Wenn die höchstwertige Ziffer, die zu unterdrücken ist, gleich 5 oder größer ist, addiere 1 zu den verbleibenden Stellen.

Das folgende Programm wird eine vierstellige BCD-Zahl in den Speicherplätzen 0040 und 0041 (MSDs in 0041) auf eine zweistellige BCD-Zahl im Speicherplatz 0041 aufrunden:

```
        LDA    $40       ;IST DAS ZU UNTERDRÜCKENDE BYTE 50
                         ;  ODER MEHR?
        CMP    #$50
        BCC    DONE
        SED              ;JA RUNDE MSDS UM 1 IN DEZIMAL
                         ;  AUF
        LDA    $41
        ADC    #0        ;ADDIERE IN ÜBERTRAG (ZU SETZEN)
        STA    $41
        CLD              ;KEHRE ZU BINÄR-BETRIEBSART ZURÜCK
DONE    BRK
```

Erinnern Sie sich daran, daß Sie den INC-Befehl nicht zur Addition von 1 verwenden können, da dieser Befehl immer ein binäres Ergebnis liefert. Der Befehl ADC #0 wird 1 zum Akkumulator addieren, da der Übertrag 1 vor der Ausführung des Befehles gesetzt werden muß (andernfalls würde der BCC-Befehl eine Verzweigung erzwungen haben). Wie gewöhnlich müssen Sie sehr sorgfältig das Flag für die Dezimal-Bertriebsart entsprechend setzen. Für größere Zahlen muß die Aufrundung durch die höchstwertigen Bits entsprechend fortgesetzt werden.

AUFGABEN:

1) Binäre Subtraktion mit mehrfacher Genauigkeit

Zweck: Subtrahiere eine Mehrwort-Zahl von einer anderen. Die Länge der Zahlen befindet sich im Speicherplatz 0040, die Zahlen selbst beginnen (höchstwertige Bits zuerst) in den Speicherplätzen 0041 und 0051, und die Differenz ersetzt die Zahl, die im Speicherplatz 0041 beginnt. Subtrahiere die Zahl, die in 0051 beginnt von der Zahl, die in 0041 beginnt.

Beispiel:

(0040) = 04

(0041) = 2F
(0042) = 5B
(0043) = A7
(0044) = C3

(0051) = 14
(0052) = DF
(0053) = 35
(0054) = B8

Ergebnis:
(0041) = 1A
(0042) = 7C
(0043) = 72
(0044) = 0B

das heißt:
2F5BA7C3
– 14DF35B8
1A7C720B

2) Dezimale Subtraktion

Zweck: Subtrahiere eine Mehrwort-Dezimalzahl (BCD) von einer anderen. Die Länge der Zahlen befindet sich in Speicherplatz 0040, die Zahlen selbst starten (höchstwertige Bits zuerst) entsprechend in den Speicherplätzen 0041 und 0051, und die Differenz ersetzt die Zahl, die im Speicherplatz 0041 beginnt. Subtrahiere die Zahl, die in 0051 beginnt, von der Zahl die in 0041 beginnt.

Beispiel:

(0040) = 04

(0041) = 36
(0042) = 70
(0043) = 19
(0044) = 85

(0051) = 12
(0052) = 66
(0053) = 34
(0054) = 59

Ergebnis:
(0041) = 24
(0042) = 03
(0043) = 85
(0044) = 26

das heißt:
36701985
– 12663459
24038526

3) Binär-Multiplikation 8 Bit mal 16 Bit

Zweck: Multipliziere die 16-Bit-Zahl ohne Vorzeichen in den Speicherplätzen 0040 und 0041 (höchstwertige Bits in 0041) mit der 8-Bit-Zahl ohne Vorzeichen im Speicherplatz 0042. Speichere das Ergebnis in die Speicherplätze 0043 bis 0045, mit den höchstwertigen Bits im Speicherplatz 0045.

Beispiele:

a.
(0040) = 03
(0041) = 00
(0042) = 05

Ergebnis:
(0043) = 0F
(0044) = 00
(0045) = 00

das heißt: 3 x 5 = 15

b.
(0040) = 6F
(0041) = 72 (29 295 dezimal)
(0042) = 61 (97 dezimal)

Ergebnis:
(0043) = 0F
(0044) = 5C
(0045) = 2B

das heißt: 29 295 x 97 = 2 841 615

4) Binäre Division mit Vorzeichen

Zweck: Dividiere die 16-Bit-Binärzahl mit Vorzeichen in den Speicherplätzen 0040 und 0041 (höchstwertige Bits in 0041) durch die 8-Bit-Zahl mit Vorzeichen im Speicherplatz 0042. Die Zahlen sind nominiert, so daß die Größe des Speicherplatzes 0042 größer als die Größe des Speicherplatzes 0041 ist. Speichere den Quotienten (mit Vorzeichen) in den Speicherplatz 0043 und den Rest (immer positiv) in den Speicherplatz 0044.

Beispiele:

a. (0040) = C0
(0041) = FF (−64)
(0042) = 08

Ergebnis: (0043) = F8 (−8) Quotient
(0044) = 00 (0) Rest

b. (0040) = 93
(0041) = ED (−4717)
(0042) = 47 (71 dezimal)

Ergebnis: (0043) = BD (−67 dezimal)
(0044) = 28 (+40 dezimal)

Hinweis: Bestimme das Vorzeichen des Ergebnisses, führe eine Division ohne Vorzeichen aus, und korrigiere den Quotienten und den Rest entsprechend.

5) Selbstprüfende Zahlen (Aligned 1, 3, 7 Mod 10)

Zweck: Berechne eine Prüfsummen-Ziffer aus einer Reihe von BCD-Ziffern. Die Länge der Reihe der Ziffern (Anzahl der Worte) befindet sich im Speicherplatz 0041. Die Ziffernreihe (2 BCD-Stellen in einem Wort) beginnt im Speicherplatz 0042. Berechne die Prüfsummen-Ziffer mit dem Verfahren "Aligned 1, 3, 7 Mod 10" und speichere sie in den Speicherplatz 0040.

Das Verfahren "Aligned 1, 3, 7 Mod 10" arbeitet wie folgt:

1) Lösche zu Beginn die Prüfsumme.
2) Addiere die führende Stelle zur Prüfsumme.
3) Multipliziere die nächste Stelle mit 3 und addiere das Ergebnis zur Prüfsumme.
4) Multipliziere die nächste Stelle mit 7 und addiere das Ergebnis zur Prüfsumme.
5) Setze das Verfahren fort (Schritt 2 bis 4) bis alle Stellen verwendet wurden.
6) Die selbstprüfende Ziffer ist die niedrigstwertige Stelle der Prüfsumme.

Beispielsweise, wenn die Ziffernreihe

549321

lautet, wird das Ergebnis sein:

Prüfsumme = 5 + 3 x 4 + 7 x 9 + 3 + 3 x 2 + 7 x 1 = 96
Selbstprüfende Ziffer = 6

Beispiele:

a. (0041) = 03
(0042) = 36
(0043) = 68
(0044) = 51

Ergebnis: Prüfsumme = 3 + 3 x 6 + 7 x 6 + 8 + 3 x 5 + 7 x 1 = 93
(0040) = 03

(0041) = 04
(0042) = 50
(0043) = 29
(0044) = 16
(0045) = 83

Ergebnis: Prüfsumme = 5 + 3 x 0 + 7 x 2 + 9 + 3 x 1 + 7 x 6 + 8 + 3 x 3 = 90
(0040) = 00

Hinweis: Beachten Sie bitte, daß 7 = 2 x 3 + 1 und 3 = 2 x 1 + 1, so daß die Formel $M_i = 2 \times M_i - 1 + 1$ verwendet werden kann, um den nächsten Multiplikationsfaktor zu berechnen.

LITERATUR

1) J. R. Herr, "Self-Checking Number Systems," Computer Design, June 1974, pp. 85-91.

2) Andere Verfahren zur Ausführung von Multiplikationen, Divisionen und anderen arithmetischen Aufgaben sind besprochen in:

   S. Davis, "Digital Processing Gets a Boost from Bipolar LSI Multipliers," EDN, November 5, 1978, pp. 38-43.

   A. Kolodzinski and D. Wainland, "Multiplying with a Microcomputer," Electronic Design, January 18, 1978, pp. 78-83.

   B. Parasuraman, "Hardware Multiplication Techniques for Microprocessor Systems," Computer Design, April 1977, pp. 75-82.

   T. F. Tao et al., "Applications of Microprocessors in Control Problems," 1977, Joint Automatic Control Conference Proceeding, San Francisco, CA., June 22-24, 1977.

   S. Waser, "State-of-the-art in High-Speed Arithmetic Integrated Circuits," Computer Design, July 1978, pp. 67-75.

   S. Waser, "Dedicated Multiplier ICs Speed Up Processing in Fast Computer Systems," Electronic Design, September 13, 1978, pp. 98-103.

   S. Waser and A. Peterson, "Medium-Speed Multipliers Trim Cost, Shrink Band- width in Speech Transmission," Electronic Design, February 1, 1979, pp. 58-65.

   A. J. Weissberger and T. Toal, "Tough Mathematical Tasks Are Child's Play for Number Cruncher," Electronics, February 17, 1977, pp. 102-107.

Kapitel 9
TABELLEN UND LISTEN

Tabellen und Listen sind zwei der grundlegenden Datenstrukturen, die bei allen Computern verwendet werden. Wir haben bereits Tabellen gesehen, die zur Ausführung von Code-Umwandlungen und arithmetischen Aufgaben eingesetzt wurden. Tabellen können auch zur Identifizierung und Ausführung von Kommandos und Befehlen, zur Linearisierung von Daten, für den Zugriff auf Dateien oder Aufzeichnungen, Definition der Bedeutung von Tasten oder Schaltern und zur Auswahl verschiedener Programme verwendet werden. Listen sind gewöhnlich weniger strukturiert als Tabellen. Listen können Aufgaben enthalten, die der Prozessor ausführen muß, Nachrichten oder Daten, die der Prozessor aufzeichnen muß, oder Bedingungen, die sich geändert haben oder überwacht werden sollten. Tabellen sind ein einfaches Hilfsmittel für das Fällen von Entscheidungen oder Lösung von Aufgaben, da keine Berechnungen oder logische Funktionen erforderlich sind. Die Aufgabe wird dann auf die Organisation der Tabelle reduziert, damit die entsprechende Eingabe leicht zu finden ist. Listen gestatten die Ausführung von Sequenzen von Aufgaben, die Vorbereitung mehrerer Resultate und den Aufbau voneinander abhängiger Dateien (oder Daten-Basen). Die Aufgaben beinhalten auch etwa das Hinzufügen von Elementen zu einer Liste oder die Entfernung von Elementen hiervon.

BEISPIELE

Hinzufügen einer Eingabe zu einer Liste

Zweck: Addiere den Inhalt des Speicherplatzes 0040 zu einer Liste, wenn er noch nicht in der Liste vorliegt. Die Länge der Liste befindet sich im Speicherplatz 0041 und die Liste selbst beginnt im Speicherplatz 0042.

Beispiele:

a.
(0040) = 6B
(0041) = 04
(0042) = 37
(0043) = 61
(0044) = 38
(0045) = 1D

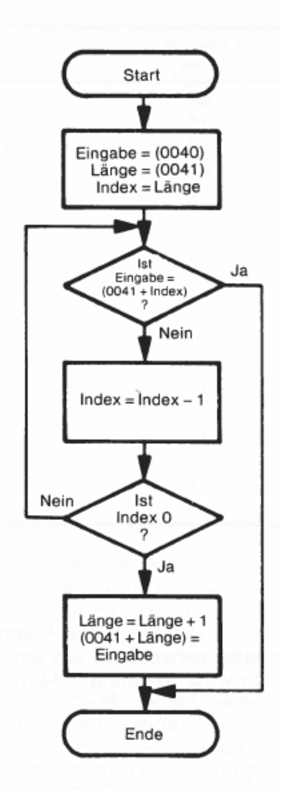
Ergebnis:
(0041) = 05
(0046) = 6B

Die Eingabe (6B) wird zur Liste hinzugefügt, da sie noch nicht vorliegt. Die Län-Liste wird um 1 erhöht.

b.
(0040) = 6B
(0041) = 04
(0042) = 37
(0043) = 6B
(0044) = 38
(0045) = 1D

Ergebnis: Keine Änderung, da die Eingabe (6B) bereits in der Liste (im Speicherplatz 0043) vorhanden ist.

Flußdiagramm:



Quellprogramm:

```
        LDA   $40     ;HOLE EINGABE
        LDX   $41     ;INDEX = LÄNGE DER LISTE
SRLST   CMP   $41,X   ;IST EINGABE = ELEMENT IN LISTE?
        BEQ   DONE    ;JA, ZU DONE
        DEX           ;NEIN, GEHE ZU NÄCHSTEM ELEMENT
                      ;  WEITER
        BNE   SRLST
        INC   $41     ;ADDIERE 1 ZUR LISTENLÄNGE
        LDX   $41
        STA   $41,X   ;ADDIERE EINGABE ZUR LISTE
DONE    BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
0000	A5		LDA	$40
0001	40			
0002	A6		LDX	$41
0003	41			
0004	D5	SRLST	CMP	$41,X
0005	41			
0006	F0		BEQ	DONE
0007	09			
0008	CA		DEX	
0009	D0		BNE	SRLST
000A	F9			
000B	E6		INC	$41
000C	41			
000D	A6		LDX	$41
000E	41			
000F	95		STA	$41,X
0010	41			
0011	00	DONE	BRK	

HASHING

Diese Methode der Addition von Elementen ist offensichtlich sehr ineffizient, wenn die Liste lang ist. Wir könnten das Verfahren verbessern, indem wir das Suchen auf einen Teil der Liste beschränken oder die Liste entsprechend anordnen. **Wir könnten das Suchen auch begrenzen, indem wir die Eingabe verwenden, um einen Startpunkt in der Liste zu erhalten. Dieses Verfahren wird "Hashing" (deutsch etwa "Zerhacken") genannt** und ähnelt sehr der Auswahl einer "Startseite" in einem Wörterbuch oder Telefonbuch mit dem ersten Buchstaben einer Eingabe. Wir könnten die Liste nach numerischen Werten anordnen. Der Suchvorgang sollte dann enden, wenn die Listenwerte unter der Eingabe liegen (größer oder kleiner, abhängig von der verwendeten Ordnungstechnik). Eine neue Eingabe müßte entsprechend eingesetzt werden, und alle anderen Eingaben müßten in der Liste nach unten verschoben werden.

Das Programm könnte so umgebaut werden, daß zwei Tabellen verwendet werden. Eine Tabelle könnte einen Startpunkt in der anderen Tabelle liefern. Zum Beispiel könnte der Suchpunkt auf der höchst- oder niedrigstwertigen 4-Bit-Ziffer in der Eingabe basieren.

Das Programm arbeitet nicht, wenn die Länge der Liste null wäre (was geschieht?). Wir können dieses Problem vermeiden, indem wir anfangs die Länge der Liste prüfen. Der Initialisierungs-Vorgang wäre dann

```
        LDX   $41        ;INDEX = LÄNGE DER LISTE
        BEQ   ADELM      ;ADDIERE EINGABE ZUR LISTE, WENN DIE
                         ;  LÄNGE NULL IST
        .
        .
        .
ADELM   INC   $41        ;ADDIERE 1 ZUR LÄNGE
```


Anders als bei einigen anderen Prozessoren wird das Null-Flag des 6502 durch Transfer-Befehle wie Laden und Speichern beeinflußt.

Wenn jede Eingabe länger als ein Wort wäre, so wäre ein Programm erforderlich, das die Bitmuster in Übereinstimmung bringt. Das Programm müßte zur nächsten Eingabe weitergehen, wenn keine Übereinstimmung auftritt, d.h. über den letzten Teil der momentanen Eingabe springen, sobald eine Nicht-Übereinstimmung gefunden wurde.

Prüfen einer geordneten Liste

Zweck: Prüfe den Inhalt des Speicherplatzes 0041 um festzustellen, ob dieser Wert in einer geordneten Liste liegt. Die Länge der Liste liegt im Speicherplatz 0042, die Liste selbst beginnt im Speicherplatz 0043 und besteht aus Binärzahlen ohne Vorzeichen in steigender Reihenfolge. Wenn der Inhalt des Speicherplatzes 0041 in der Liste liegt, lösche Speicherplatz 0040. Andernfalls setze Speicherplatz 0040 auf FF_{16}.

Beispiele:

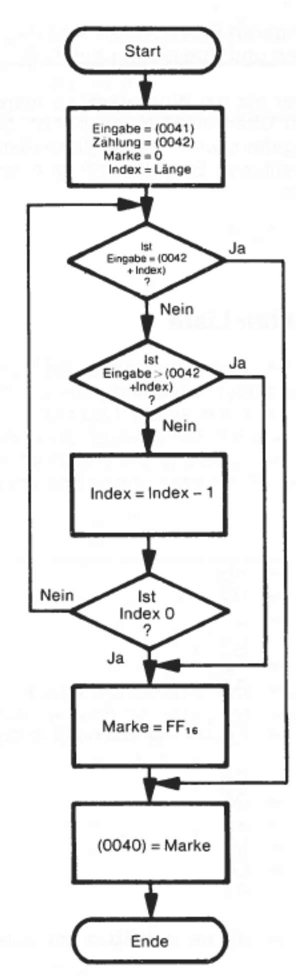
a.
(0041) = 6B
(0042) = 04
(0043) = 37
(0044) = 55
(0045) = 7D
(0046) = A1

Ergebnis: (0040) = FF, da sich 6B nicht in der Liste befindet.

b.
(0041) = 6B
(0042) = 04
(0043) = 37
(0044) = 55
(0045) = 6B
(0046) = A1

Ergebnis: (0040) = 00, da sich 6B in der Liste befindet.

Flußdiagramm:



Das Suchverfahren ist ein wenig anders, da hier die Elemente geordnet sind. Sobald wir ein Element finden, das kleiner ist als die Eingabe (erinnern Sie sich daran, daß wir uns rückwärts durch die Liste in der gebräuchlichen Art und Weise des 6502 bewegen), das Suchen vorüber ist, da weiter folgende Elemente noch kleiner sein werden. Sie könnten vielleicht ein Beispiel versuchen, um sich zu überzeugen, daß dieses Verfahren arbeitet. Beachten Sie, daß ein Element kleiner als die Eingabe durch einen Vergleich angezeigt wird, der kein "Borgen" (d.h. Übertrag = 1) erzeugt.

SUCH-METHODEN

Wie bei der vorhergehenden Aufgabe könnte eine Tabelle oder ein anderes Verfahren einen guten Startpunkt wählen, um das Suchen zu beschleunigen. **Ein Verfahren bestünde im Starten in der Mitte und im Bestimmen, in welcher Hälfte der Liste die Eingabe war, dann die Hälfte in Hälften zu teilen etc. Dieses Verfahren wird binäres Suchen benannt, da es den verbleibenden Rest jedesmal in die Hälfte teilt.**[1]

Quellprogramm:

```
        LDA  $41      ;HOLE EINGABE
        LDX  $42      ;INDEX = LÄNGE DER LISTE
        LDY  #0       ;MARKE = NULL FÜR ELEMENT IN LISTE
SRLST   CMP  $42,X    ;IST EINGABE GLEICH MIT ELEMENT?
        BEQ  DONE     ;JA, SUCHEN ABGESCHLOSSEN
        BCS  NOTIN    ;EINGABE NICHT IN LISTE, WENN
                      ;   GRÖSSER ALS ELEMENT
        DEX
        BNE  SRLST
NOTIN   LDY  #$FF     ;MARKE = FF FÜR NICHT IN DER LISTE
DONE    STY  $40      ;BEWAHRE MARKE AUF
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A5		LDA	$41
0001	41			
0002	A6		LDX	$42
0003	42			
0004	A0		LDY	#0
0005	00			
0006	D5	SRLST	CMP	$42,X
0007	42			
0008	F0		BEQ	DONE
0009	07			
000A	B0		BCS	NOTIN
000B	03			
000C	CA		DEX	
000D	D0		BNE	SRLST
000E	F7			
000F	A0	NOTIN	LDY	#$FF
0010	FF			
0011	84	DONE	STY	$40
0012	40			
0013	00		BRK	

Dieser Algorithmus ist ein wenig langsamer, als jener in dem Beispiel, das in "Hinzufügen einer Eingabe zu einer Liste" angegeben wurde, infolge des zusätzlichen bedingten Sprunges (BCS NOTIN). Die durchschnittliche Ausführungszeit für dieses einfache Suchverfahren steigt linear mit der Länge der Liste an, während die durchschnittliche Ausführungszeit für einen binären Suchvorgang logarithmisch ansteigt. Wenn zum Beispiel die Länge der Liste doppelt so groß ist, so benötigt das einfache Verfahren doppelt so lang wie der Durchschnitt, während das binäre Suchen nur eine zusätzliche Wiederholung erfordert.

Entfernen eines Elementes von einer Warteschlange

Zweck: Die Speicherplätze 0042 und 0043 enthalten die Adresse der Spitze (Kopf) der Warteschlange (MSBs in 0043). Plaziere die Adresse des ersten Elementes (Kopf) der Warteschlange in die Speicherplätze 0040 und 0041 (MSBs in 0041) und bringe die Warteschlange auf den neuesten Stand, um das Element zu entfernen. Jedes Element in der Schlange ist zwei Bytes lang und enthält die Adresse des nächsten 2-Byte-Elementes in der Schlange. Das letzte Element in der Warteschlange enthält null, um anzuzeigen, daß es kein weiteres Element gibt.

Warteschlangen werden zum Speichern von Daten in der Reihenfolge verwendet, in der sie gebraucht werden, oder für Aufgaben in der Reihenfolge, in der sie ausgeführt werden. Die Warteschlange besitzt eine "Zuerst-ein-, Zuerst-aus-"-Datenstruktur, d.h. Elemente werden von der Warteschlange in der gleichen Reihenfolge entfernt, in der sie eingegeben wurden. Betriebs-Systeme plazieren Aufgaben in Warteschlangen, so daß sie in der entsprechenden Reihenfolge ausgeführt werden. E/A-Treiber transferieren Daten zu oder von Warteschlangen, so daß sie übertragen oder in der entsprechenden Reihenfolge verarbeitet werden. Puffer können ebenfalls in Warteschlangen angeordnet werden, so daß der nächste verfügbare Puffer leicht gefunden werden kann, und jene, die ausgegeben wurden, leicht zum jeweils verfügbaren Speicher hinzugefügt werden können. Warteschlangen können auch zur Aneinander-Reihung von Anforderungen für Speicherung, zeitliche Steuerung oder E/A verwendet werden, so daß diese in der korrekten Reihenfolge befriedigt werden können.

In realen Anwendungen wird jedes Element in der Warteschlange typisch eine große Anzahl von Informationen oder Speicherraum neben den erforderlichen Adressen enthalten, damit ein Element mit dem nächsten verbunden werden kann.

Beispiele:

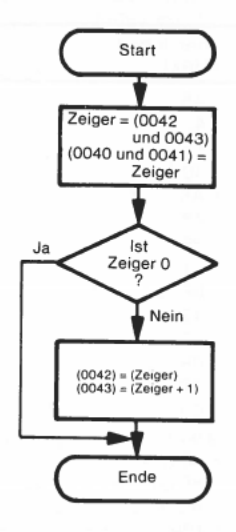
```
a.          (0042) = 46 } Adresse des ersten Elementes
            (0043) = 00 } in der Warteschlange
            (0046) = 4D } Adresse des zweiten Elementes
            (0047) = 00 } in der Warteschlange
            (004D) = 00 } Ende der Warteschlange
            (004E) = 00 }

Ergebnis:   (0040) = 46 } Adresse des aus der Warteschlange
            (0041) = 00 } entfernten Elementes
            (0042) = 4D } Adresse des neuen ersten
            (0043) = 00 } Elementes in der Warteschlange

b.          (0042) = 00 } Leere Warteschlange
            (0043) = 00 }

Ergebnis:   (0040) = 00 } Kein Element von der
            (0041) = 00 } Warteschlange verfügbar
```


Flußdiagramm:



Quellprogramm:

```
        LDA   $42      ;ENTFERNE KOPF VON WARTESCHLANGE
        STA   $40
        LDA   $43
        STA   $41
        ORA   $42      ;IST WARTESCHLANGE LEER?
        BEQ   DONE     ;JA, DONE
        LDY   #0       ;NEIN, BRINGE NÄCHSTES ELEMENT ZUM
                       ;  KOPF DER WARTESCHLANGE
        LDA   ($40),Y
        STA   $42
        INY
        LDA   ($40),Y
        STA   $43
DONE    BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
0000	A5		LDA	$42
0001	42			
0002	85		STA	$40
0003	40			
0004	A5		LDA	$43
0005	43			
0006	85		STA	$41
0007	41			
0008	05		ORA	$42
0009	42			
000A	F0		BEQ	DONE
000B	0B			
000C	A0		LDY	#0
000D	00			
000E	B1		LDA	($40).Y
000F	40			
0010	85		STA	$42
0011	42			
0012	C8		INY	
0013	B1		LDA	($40).Y
0014	40			
0015	85		STA	$43
0016	43			
0017	00	DONE	BRK	

Mit dem Anordnen in Warteschlangen kann man Listen verarbeiten, die sich nicht in aufeinanderfolgenden Speicherplätzen befinden. Jedes Element in der Warteschlange muß die Adresse des nächsten Elementes enthalten. Derartige Listen gestatten Ihnen, Daten oder Aufgaben in der entsprechenden Reihenfolge zu verarbeiten, Variable zu ändern oder Definitionen in ein Programm einzufügen. Hierfür ist zusätzlicher Speicherplatz erforderlich, jedoch können die Elemente leicht hinzugefügt oder entfernt werden.

Nach-indizierte oder indirekte indizierte Adressierung ist hier sehr handlich, da es uns gestattet, den Inhalt der Speicherplätze 0040 und 0041 als Zeiger zu verwenden. Diese Speicherplätze enthalten die Adresse des Kopfes der Warteschlange, der wiederum die Adresse des nächsten Elementes enthält. Die Speicherplätze, in welchen die Adresse des Elementes gespeichert ist, muß auf Seite null sein, da sie mit nach-indizierter Adressierung verwendet werden. Alle anderen Adressen können irgendwo im Speicher liegen. Die nach-indizierte Adressierung könnte ebenfalls später verwendet werden, um Daten zu oder von dem Element zu transferieren, das eben von der Warteschlange entfernt worden ist.

Erinnern Sie sich daran, daß Nach-Indizierung nur für Adressen auf der Nullseite verfügbar ist. Ferner kann nur das Indexregister Y in dieser Betriebsart verwendet werden.

Beachten Sie die Verwendung der Sequenz

```
LDA  $43
ORA  $42
```

zur Bestimmung, ob die 16-Bit-Zahl in den Speicherplätzen 0042 und 0043 gleich null ist. Versuchen Sie einige andere Sequenzen zu finden, die diese Aufgabe handhaben könnten – es tritt offensichtlich auf, wann immer Sie einen 16-Bit-Zähler anstatt des 8-Bit-Zählers einsetzen, den wir in den meisten der Beispiele verwendet haben.

Ein Problem mit dem Befehlssatz des 6502 besteht darin, daß er keine Befehle hat, die speziell 16-Bit-Adressen (oder Daten) von einer Stelle zu einer anderen transferieren oder andere 16-Bit-Operationen ausführen können. Natürlich müßten derartige Befehle jeweils 8 Bits gleichzeitig bearbeiten, es könnten jedoch etliche Zyklen für das Holen und Decodieren des Befehls eingespart werden. Die meisten übrigen Mikroprozessoren besitzen derartige Befehle.

Es kann sehr nützlich sein, Zeiger an beiden Enden der Warteschlange einzurichten, anstatt nur an deren Kopf.[2,3] Die Datenstruktur kann dann entweder in einer "Zuerst-ein-, Zuerst-aus"-Weise oder in einer "Zuletzt-ein-, Zuerst-aus"-Weise verwendet werden, abhängig davon, ob neue Elemente zum Kopf oder zum Schwanz der Warteschlange hinzugefügt werden. Wie würden Sie das Programmbeispiel ändern, damit die Speicherplätze 0044 und 0045 die Adresse des letzten Elementes (Schwanz) der Warteschlange enthalten?

Wenn es keine Elemente in der Warteschlange gibt, so löscht das Programm die Speicherplätze 0040 und 0041. Ein Programm, das ein Element aus der Warteschlange anfordert, würde dann diese Speicherplätze zu prüfen haben, um festzustellen, ob die Anfrage erledigt wurde. Können Sie sich andere Arten vorstellen, mit denen diese Information zu erhalten ist?

8-Bit-Sortierung

Zweck: Sortiere eine Anordnung von Binärzahlen ohne Vorzeichen in abnehmender Reihenfolge. Die Länge der Anordnung liegt im Speicherplatz 0040, und die Anordnung selbst beginnt im Speicherplatz 0041.

Beispiel:

(0040) = 06
(0041) = 2A
(0042) = B5
(0043) = 60
(0044) = 3F
(0045) = D1
(0046) = 19

Ergebnis:

(0041) = D1
(0042) = B5
(0043) = 60
(0044) = 3F
(0045) = 2A
(0046) = 19

Eine einfache Sortier-Technik arbeitet folgendermaßen:

EINFACHER SORTIER-ALGORITHMUS

Schritt 1) Lösche ein Flag INTER.
Schritt 2) Prüfe jedes aufeinanderfolgende Paar von Zahlen in der Anordnung. Wenn sich hiervon welche nicht in der Reihenfolge befinden, tausche sie aus und setze INTER.
Schritt 3) Wenn INTER = 1, nachdem die gesamte Anordnung geprüft wurde, gehe zu Schritt 1 zurück.

INTER wird gesetzt, wenn irgendein aufeinanderfolgendes Paar von Zahlen nicht in der Reihenfolge liegt. Daher befindet sich, wenn INTER = 0 am Ende eines Durchlaufes durch die gesamte Anordnung, diese Anordnung in der richtigen Reihenfolge.

Dieses Sortier-Verfahren wird als "Bubble sort" (deutsch etwa "stufenweise Sortierung") bezeichnet. Der Algorithmus ist leicht auszuführen. Es sollten jedoch andere Sortier-Verfahren untersucht werden, wenn lange Listen zu sortieren sind und die Geschwindigkeit von Bedeutung ist.[1,4,5]

Diese Technik arbeitet in einem einfachen Fall wie folgt. Nehmen wir an, daß wir eine Reihe in abnehmender Reihenfolge sortieren wollen. Die Reihe besitzt vier Elemente – 12, 03, 15, 08. Wir wollen uns rückwärts durch die Anordnung arbeiten, wie es für den 6502 üblich ist.

Erste Wiederholung:

Schritt 1) INTER = 0

Schritt 2) Abschließende Reihenfolge der Reihe ist:
15
12
03
08
da das zweite Paar (03, 15) ausgetauscht wurde und ebenso das dritte Paar (12, 15).
INTER = 1

Zweite Wiederholung:

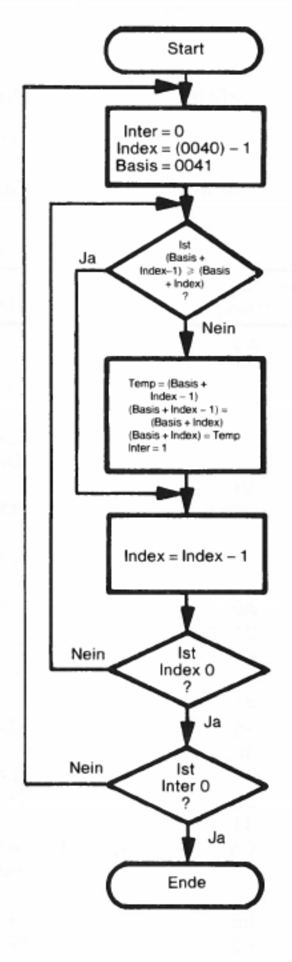
Schritt 1) INTER = 0
Schritt 2) Abschließende Anordnung dieser Reihe ist:
15
12
08
03
da das erste Paar (08, 03) ausgetauscht wurde. INTER = 1.

Dritte Wiederholung:

Schritt 1) INTER = 0
Schritt 2) die Elemente befinden sich bereits in der richtigen Reihenfolge, so daß keine Vertauschungen erforderlich sind und INTER gleich null bleibt.

Beachten Sie, daß immer eine zusätzliche Wiederholung ausgeführt wird um festzustellen, daß die Elemente in der richtigen Reihenfolge liegen. Hier sind offensichtlich noch zahlreiche Verbesserungen möglich, und neue Sortierverfahren werden laufend eingehend untersucht.

Flußdiagramm:



Quellprogramm:

```
SORT    LDY   #0       ;AUSTAUSCH-FLAG = 0
        LDX   $40      ;HOLE LÄNGE DER ANORDNUNG
        DEX            ;ORDNE REIHENLÄNGE IN ANZAHL DER
                       ;  PAARE
PASS    LDA   $40,X    ;IST DAS PAAR DER ELEMENTE IN DER
                       ;  REIHENFOLGE?
        CMP   $41,X
        BCS   COUNT    ;JA, VERSUCHE NÄCHSTES PAAR
        LDY   #1       ;NEIN, SETZE AUSTAUSCH-FLAG
        PHA            ;TAUSCHE ELEMENTE UNTER
                       ;  VERWENDUNG DES STAPELS AUS
        LDA   $41,X
        STA   $40,X
        PLA
        STA   $41,X
COUNT   DEX            ;PRÜFE AUF VOLLSTÄNDIGEN DURCHLAUF
        BNE   PASS
        DEY            ;WAREN ALLE ELEMENTE IN DER
                       ;  REIHENFOLGE?
        BEQ   SORT     ;NEIN, GEHE WIEDER DURCH DIE REIHE
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A0	SORT	LDY	#0
0001	00			
0002	A6		LDX	$40
0003	40			
0004	CA		DEX	
0005	B5	PASS	LDA	$40,X
0006	40			
0007	D5		CMP	$41,X
0008	41			
0009	B0		BCS	COUNT
000A	0A			
000B	A0		LDY	#1
000C	01			
000D	48		PHA	
000E	B5		LDA	$41,X
000F	41			
0010	95		STA	$40,X
0011	40			
0012	68		PLA	
0013	95		STA	$41,X
0014	41			
0015	CA	COUNT	DEX	
0016	D0		BNE	PASS
0017	ED			
0018	88		DEY	
0019	F0		BEQ	SORT
001A	E5			
001B	00		BRK	

Der Fall, bei dem zwei Elemente in der Anordnung gleich sind, ist sehr wichtig. Das Programm sollte einen Austausch in diesem Fall nicht ausführen, da dieser Austausch bei jedem Durchlauf erfolgen würde. Das Ergebnis wäre, daß jeder Durchlauf das Austauschflag setzen würde, und damit eine endlose Schleife zur Folge hätte. Das Programm vergleicht die Elemente in der spezifizierten Reihenfolge, so daß das Übertrags-Flag gesetzt wird, wenn die Elemente bereits korrekt angeordnet sind. Erinnern Sie sich daran, daß der Vergleich zweier gleicher Werte das Übertrags-Flag setzt, da das Flag ein invertiertes "Borgen" nach Subtraktionen oder Vergleichen ist.

Die Befehle für bedingte Verzweigungen des 6502 haben ihre Grenzen und speziell in diesem Programm. Folgen wir einem Befehl wie CMP, so haben wir nur BCC, eine Verzweigung wenn $(M) > (A)$, und BCS, verzweige wenn $(M) \leqslant (A)$. Der 6502 besitzt keine Verzweigungsbefehle für die Fälle, bei denen die Gleichheits-Bedingung auf der anderen Seite liegt, d.h.: $(M) \geqslant (A)$ und $(M) \leqslant (A)$. Deshalb müssen wir die Reihenfolge der Operationen sehr sorgfältig beachten.

Vor dem Starten jedes Sortier-Durchlaufes müssen wir darauf achten, den Index und das Austausch-Flag zu initialisieren.

Das Programm muß den Zähler um 1 zu Anfang reduzieren, da die Anzahl der aufeinander folgenden Paare um 1 kleiner ist als die Anzahl der Elemente (dem letzten Element folgt kein weiteres).

Dieses Programm arbeitet nicht ordnungsgemäß, wenn es weniger als zwei Elemente in der Anordnung gibt. Wie können Sie diesen Fall handhaben?

ANDERE SORTIER-VERFAHREN

Es gibt zahlreiche Sortier-Algorithmen, die sich in ihrer Effizienz sehr weitgehend unterscheiden. Die Literatur 1, 4 und 5 beschreiben einige hiervon.

Der Stapel ist sehr leicht für zeitweilige Speicherung in diesem Programm zu verwenden, da die Befehle PHA (Push Accumulator oder Store Accumulator in Stack) und PLA (Pull Accumulator oder Load Accumulator from Stack) nur jeweils ein Byte lang sind. Die Adresse liegt im Stapelzeiger (erweitert mit 01 wie seine Seitennummer). Wenn Sie wollen, können Sie einen festen Speicherplatz ersetzen, wie etwa 003F. Der Austausch ist dann:

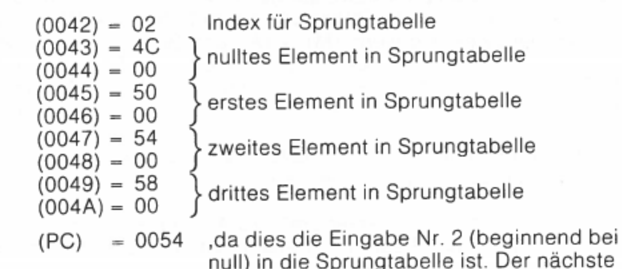
```
STA   $3F      ;TAUSCHE ELEMENTE UNTER VERWEN-
               ;  DUNG EINER ZWISCHENSPEICHERUNG
               AUS
LDA   $41,X
STA   $40,X
LDA   $3F
STA   $41,X
```

Siehe Kapitel 10 für eine weitere Besprechung des RAM-Stapels des 6502.

Verwendung einer geordneten Sprungtabelle

Zweck: Verwende den Inhalt des Speicherplatzes 0042 als einen Index zu einer Sprungtabelle, beginnend im Speicherplatz 0043. Jede Eingabe in die Sprungtabelle enthält eine 16-Bit-Adresse mit den LSBs im ersten Byte. Das Programm sollte die Steuerung zu der Adresse mit dem entsprechenden Index transferieren, d.h. wenn der Index 6 ist, springt das Programm zur Adressen-Eingabe Nr. 6 in der Tabelle. Nimm an, daß die Tabelle weniger als 128 Eingaben besitzt.

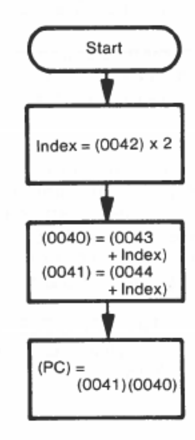
Beispiel:



(0042) = 02 Index für Sprungtabelle
(0043) = 4C
(0044) = 00 } nulltes Element in Sprungtabelle
(0045) = 50
(0046) = 00 } erstes Element in Sprungtabelle
(0047) = 54
(0048) = 00 } zweites Element in Sprungtabelle
(0049) = 58
(004A) = 00 } drittes Element in Sprungtabelle

Ergebnis: (PC) = 0054 ,da dies die Eingabe Nr. 2 (beginnend bei null) in die Sprungtabelle ist. Der nächste auszuführende Befehl wird der eine sein, der an dieser Stelle liegt.

Flußdiagramm:



Quellprogramm:

```
LDA   $42      ;HOLE INDEX
ASL   A        ;VERDOPPLE INDEX FÜR 2-BYTE-TABELLE
TAX
LDA   $43,X    ;HOLE LSB'S DER SPRUNGADRESSE
STA   $40
LDA   $44,X    ;HOLE MSB'S DER SPRUNGADRESSE
STA   $41
JMP   ($40)    ;TRANSFERIERE STEUERUNG ZU
               ;  BESTIMMUNGSORT
```

Das letzte Kästchen resultiert in einem Transfer der Steuerung zur Adresse, die aus der Tabelle erhalten wird.

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)	
0000	A5	LDA	$42
0001	42		
0002	0A	ASL	A
0003	AA	TAX	
0004	B5	LDA	$43,X
0005	43		
0006	85	STA	$40
0007	40		
0008	B5	LDA	$44,X
0009	44		
000A	85	STA	$41
000B	41		
000C	6C	JMP	($40)
000D	40		
000E	00		

Sprungtabellen sind in Situationen sehr nützlich, in denen eine von mehreren Routinen ausgewählt werden muß. Derartige Situationen treten beim Decodieren von Kommandos (beispielsweise eingegeben von einer Tastatur), Auswahl von Testprogrammen, Wahl unterschiedlicher Methoden oder bei der Auswahl einer E/A-Konfiguration auf.

Die Sprungtabelle ersetzt eine ganze Serie von bedingten Sprung-Operationen. Das Programm, das auf die Sprungtabelle zugreift, könnte zum Zugriff auf mehrere unterschiedliche Tabellen verwendet werden, indem einfach die nach-indizierte oder indirekte indizierte Adressier-Art verwendet wird, bei der die Start-Adresse der Tabelle in das RAM auf Seite null plaziert wird.

Die Daten müssen mit 2 multipliziert werden, um den richtigen Index zu ergeben, da jede Eingabe in der Sprungtabelle zwei Bytes belegt.

Der Befehl JMP ($40) verwendet indirekte Adressierung. Der Bestimmungsort ist die Adresse, die in die spezifizierte Speicherstelle anstatt die spezifizierte Speicherstelle selbst gespeichert ist. JMP ist der einzige Befehl des 6502, der indirekte Adressierung verwendet. Beachten Sie, daß es keine Nullseiten-Betriebsart gibt und daß die Adresse auf die gebräuchliche Art des 6502 gespeichert wird, mit den niedrigstwertigen Bits zuerst.

SPRUNG- UND VERZWEIGUNGS-TERMINOLOGIE

Die Terminologie, die bei der Beschreibung von Sprung- oder Verzweigungsbefehlen verwendet wird, ist häufig verwirrend. Ein Sprungbefehl, der mit Verwendung direkter Adressierung beschrieben wird, lädt in Wirklichkeit die spezifizierte Adresse in den Befehlszähler. Dies arbeitet mehr wie unmittelbare Adressierung als direkte Adressierung, wie sie bei anderen Befehlen wie Laden oder Speichern angewendet wird. Ein Sprungbefehl unter Verwendung indirekter Adressierung arbeitet wie ein anderer Befehl mit Verwendung direkter Adressierung.

Es ist keine Abschluß-Operation erforderlich (wie ein BRK-Befehl), da JMP ($40) die Steuerung zu der Adresse transferiert, die von der Sprungtabelle enthalten wird.

Literatur 7 und 8 enthalten zusätzliche Beispiele für die Verwendung von Sprungtabellen. Das Programm nimmt an, daß die Sprungtabelle weniger als 128 Einheiten enthält (weshalb?). Wie könnten Sie das Programm ändern, um längere Tabellen zu ermöglichen?

AUFGABEN

1) Entfernen einer Eingabe aus einer Liste

Zweck: Entferne den Inhalt des Speicherplatzes 0040 aus einer Liste, wenn dieser vorliegt. Die Länge der Liste befindet sich im Speicherplatz 0041, und die Liste selbst beginnt im Speicherplatz 0042. Bringe die Eingaben unter die eine, die um eine Stelle nach oben bewegt wurde, und verringere die Länge der Liste um 1.

Beispiele:

a. (0040) = 6B aus der Liste zu entfernende Eingabe

(0041) = 04 Länge der Liste

(0042) = 37 erstes Element in der Liste
(0043) = 61
(0044) = 28
(0045) = 1D

Ergebnis: Keine Änderung, da die Eingabe nicht in der Liste ist.

b. (0040) = 6B aus der Liste zu entfernende Eingabe

(0041) = 04 Länge der Liste

(0042) = 37 erstes Element in der Liste
(0043) = 6B
(0044) = 28
(0045) = 1D

Ergebnis: (0041) = 03 Länge der Liste wird um 1 vermindert

(0042) = 37
(0043) = 28 die anderen Elemente in der Liste werden um 1 Position nach oben verschoben
(0044) = 1D

Die Eingabe wird aus der Liste entfernt, und die eine unterhalb wird um eine Position nach oben verschoben. Die Länge der Liste wird um 1 verringert.

2) Hinzufügen einer Eingabe zu einer geordneten Liste

Zweck: Plaziere den Inhalt des Speicherplatzes 0040 in eine geordnete Liste, wenn sie nicht bereits vorhanden ist. Die Länge der Liste liegt im Speicherplatz 0041, und die Liste selbst beginnt im Speicherplatz 0042 und besteht aus Binärzahlen ohne Vorzeichen in steigender Reihenfolge. Plaziere die neue Eingabe in die richtige Position in der Liste, ordne die Elemente unter dieser an, und erhöhe die Länge der Liste um 1.

Beispiele:

a. (0040) = 6B zur Liste hinzuzufügende Eingabe

(0041) = 04 Länge der Liste

(0042) = 37 erstes Element in der Liste
(0043) = 55
(0044) = 7D
(0045) = A1

Ergebnis: (0041) = 05 Länge der Liste wird um 1 erhöht

(0044) = 6B in die Liste plazierte Eingabe
(0045) = 7D die anderen Elemente in der Liste werden um eine Position nach unten verschoben

(0046) = A1

b. (0040) = 6B zur Liste hinzuzufügende Eingabe

(0041) = 04 Länge der Liste

(0042) = 37 erstes Element in der Liste
(0043) = 55
(0044) = 6B
(0045) = A1

Das Ergebnis besteht darin, daß keine Änderung erfolgt, da die Eingabe bereits in der Liste vorhanden ist.

3) Addieren eines Elementes zu einer Warteschlange

Zweck: Addiere die Adresse in den Speicherplätzen 0040 und 0041 (MSBs in 0041) zu einer Warteschlange. Die Adresse des ersten Elementes der Schlange liegt in den Speicherplätzen 0042 und 0043 (MSBs in 0043). Jedes Element in der Schlange enthält entweder die Adresse des nächsten Elementes in der Schlange oder null, wenn es kein nächstes Element gibt. Alle Adressen sind 16 Bits lang, mit den niedrigstwertigen Bits im ersten Byte des Elementes. Das neue Element gelangt zum Ende (Schwanz) der Schlange. Seine Adresse wird in dem Element liegen, das am Ende der Schlange lag, und wird null enthalten um anzuzeigen, daß es nun das Ende der Warteschlange ist.

Beispiel:

(0040) = 4D } Neues Element, das der Warte-
(0041) = 00 } schlange hinzuzufügen ist.
(0042) = 46 } Zeiger zum Kopf der Warteschlange
(0043) = 00 }

(0046) = 00 } Letztes Element in der Warteschlange
(0047) = 00 }

Ergebnis:

(0046) = 4D } Altes letztes Element zeigt zum
(0047) = 00 } neuen letzten Element

(004D) = 00 } Neues letztes Element in der
(004E) = 00 } Warteschlange

Wie würden Sie ein Element zur Warteschlange hinzufügen, wenn die Speicherplätze 0044 und 0045 die Adresse des Schwanzes der Warteschlange (das letzte Element) enthalten würden?

4) 16-Bit-Sortierung

Zweck: Sortiere eine Reihe von 16-Bit-Binärzahlen ohne Vorzeichen in absteigender Reihenfolge. Die Länge der Reihe befindet sich im Speicherplatz 0040, und die Reihe selbst beginnt im Speicherplatz 0041. Jede 16-Bit-Zahl wird mit den höchstwertigen Bits im ersten Byte gespeichert.

Beispiel:

(0040) = 03 Länge der Liste

(0041) = D1 LSBs des ersten Elementes in der Liste
(0042) = 19 MSBs des ersten Elementes in der Liste

(0043) = 60
(0044) = 3F

(0045) = 2A
(0046) = B5

Ergebnis:

(0041) = 2A LSBs des ersten Elementes in der geordneten Liste
(0042) = B5 MSBs des ersten Elementes in der geordneten Liste

(0043) = 60
(0044) = 3F

(0045) = D1
(0046) = 19

Die Zahlen sind B52A, 3F60 und 19D1.

5) Verwendung einer Sprungtabelle mit einem Schlüssel

Zweck: Verwende den Inhalt des Speicherplatzes 0040 als Schlüssel zu einer Sprungtabelle, die im Speicherplatz 0041 beginnt. Jede Eingabe in die Sprungtabelle enthält einen 8-Bit-Schlüsselwert, gefolgt von einer 16-Bit-Adresse (MSBs im ersten Wort), zu der das Programm die Steuerung transferieren sollte, wenn der Schlüssel gleich dem Schlüsselwert ist.

Beispiel:

(0040) = 38 Schlüssel-Wert für Suchen

(0041) = 32 Schlüssel-Wert für erste Eingabe
(0042) = 4A LSBs der Sprungadresse für erste Eingabe
(0043) = 00 MSBs der Sprungadresse für erste Eingabe

(0044) = 35
(0045) = 4E
(0046) = 00

(0047) = 38
(0048) = 52
(0049) = 00

Ergebnis:

(PC) = 005F, da die Adresse mit dem Schlüsselwert 38 übereinstimmt.

LITERATUR

1. D. Knuth, The Art of Computer Programming, Volume III: Sorting and Searching, (Reading, Mass./ Addison-Wesley, 1978).

2. K. J. Thurber and P. C. Patton, Data Structures and Computer Architecture, (Lexington, Mass.: Lexington Books, 1977).

3 J. Hemenway an E. Teja, "Data Structures – Part 1", EDN, March 5, 1979, pp. 89-92.

4. B. W. Kernighan and P. J. Plauger, The Elements of Programming Style, (New York: McGraw-Hill, 1978).

5. K. A. Schember and J. R. Rumsey, "Minimal Storage Sorting and Searching Techniques for RAM Applications", Computer, June 1977, pp. 92-100.

6. "Sorting 30 Times Faster with DPS", Datamation, February 1978, pp. 200-203.

7. L. A. Leventhal, "Cut Your Processor's Computation Time", Electronic Design, August 16, 1977, pp. 82-89.

8. J. B. Peatman, Microcomputer-Based Design, (New York: McGraw-Hill, 1977), Chapter 7.

Kapitel 10
UNTERPROGRAMME

Keiner der bisher geeigneten Beispiele stellt ein Programm für sich selbst dar. Die meisten realen Programme führen eine Serie von Aufgaben aus, und viele von ihnen können gleich sein oder gemeinsam in unterschiedlichen Programmen vorkommen. Wir benötigen daher einen Weg, um diese Aufgaben einmal zu formulieren und sie dann bequem in verschiedenen Teilen des momentanen Programmes und in anderen Programmen verfügbar zu machen.

UNTERPROGRAMM-BIBLIOTHEK

Die Standard-Methode besteht darin, Unterprogramme zu schreiben, die spezielle Aufgaben ausführen. Die sich ergebenden Sequenzen von Befehlen können einmal geschrieben, getestet und dann wiederholt werden. Sie können eine Unterprogramm-Bibliothek bilden, die dokumentierte Lösungen für allgemeine Probleme liefert.

UNTERPROGRAMM-BEFEHLE

Die meisten Mikroprozessoren besitzen spezielle Befehle für die Übergabe der Steuerung an Unterprogramme und das Zurückgeben der Steuerung zum Hauptprogramm. Wir beziehen uns häufig auf den speziellen Befehl, der die Steuerung einem Unterprogramm mittels Aufruf (Call), Springe-zu-Unterprogramm (Jump to Subroutine), Springe-und-markiere-Stelle (Jump and Mark place) oder Springe-und-verbinde (Jump and Link) übergibt. Der spezielle Befehl, der die Steuerung dem Hauptprogramm zurückgibt, wird gewöhnlich Rückkehr (Return) genannt. Beim Mikroprozessor 6502 bewahrt der Befehl "Springe zu Unterprogramm" (JSR = Jump to Subroutine) den alten Wert des Befehlszählers im RAM-Stapel auf, bevor er die Start-Adresse des Unterprogramms in den Befehlszähler plaziert. Der Befehl "Kehre von Unterprogramm zurück" (RTS = Return from Subroutine) holt den alten Wert vom Stapel und legt ihn zurück in den Befehlszähler. Der Zweck besteht im Transfer der Programmsteuerung, zuerst zum Unterprogramm und dann zurück zum Hauptprogramm. Natürlich kann das Unterprogramm selbst die Steuerung an ein weiteres Unterprogramm abgeben, usw.

Um wirklich von Nutzen zu sein, muß ein Unterprogramm universell ausgelegt werden. Eine Routine, die nur eine spezialisierte Aufgabe ausführen kann, wie etwa das Suchen nach einem bestimmten Buchstaben in einer Eingangsreihe fester Länge, wird nicht sehr nützlich sein. Wenn andererseits das Unterprogramm jeden Buchstaben in einer Reihe beliebiger Länge aufsuchen kann, so wird dies wesentlich nützlicher sein. **Wir nennen die Daten oder Adressen, die dem Unterprogramm Abänderungen gestatten, "Parameter".** Ein wesentlicher Teil beim Schreiben von Unterprogrammen besteht in der Entscheidung, welche Variablen Parameter sein sollen.

Ein Problem besteht im Transferieren der Parameter in das Unterprogramm. Dieser Vorgang wird die "Übergabe von Parametern" (passing parameters) genannt. Das einfachste Verfahren besteht für das Hauptprogramm darin, die Parameter in Register zu plazieren. Dann kann das Unterprogramm einfach annehmen, daß sich die Parameter dort befinden. Natürlich ist diese Technik durch die Anzahl der verfügbaren Register begrenzt. Die Parameter können jedoch so wie Daten adressiert werden. Beispielsweise könnte eine Sortier-Routine mit dem Index-Register X beginnen, das die Adresse auf der Null-Seite enthält, in der die Länge der Anordnung liegt.

ÜBERGABE VON PARAMETERN

Der Mikroprozessor 6502 ist durch die Tatsache begrenzt, daß er keine Register für die volle Adressenlänge (16 Bit) besitzt, in die entsprechend lange Parameter übergeben werden können. Es können jedoch derartige Parameter leicht übertragen werden, indem Speicherplätze auf Seite Null reserviert werden. Diese Speicherplätze arbeiten effektiv als zusätzliche Register. Ein weiterer Vorteil dieser Lösung besteht darin, daß auf Adressen auf der Nullseite unter Verwendung der nach-indizierten (indirekt indizierten) und vor-indizierten (indizierten indirekten) Adressierung zugegriffen werden kann, als auch mit den kurzen Nullseiten-Formen der direkten und indizierten Adressierung.

Eine weitere Möglichkeit besteht in der Verwendung des Stapels. Das Hauptprogramm kann die Parameter in den Stapel plazieren und das Unterprogramm kann sie von dort zurückgewinnen. Die Vorteile dieses Verfahrens liegen darin, daß der Stapel gewöhnlich ziemlich lang ist (bis zu einer Seite), und daß Daten im Stapel nicht verloren gehen, auch wenn der Stapel erneut verwendet wird. Der Nachteil ist darin zu sehen, daß wenige Befehle "Springe zu Unterprogramm" die Rückkehr-Adresse in den Stapel speichert.

Ein anderes Verfahren besteht in der Verwendung eines Speicherbereichs für Parameter. Das Hauptprogramm kann die Adresse dieses Bereichs in der Null-Seite plazieren, und das Unterprogramm kann die Daten, falls nötig, von dort unter Verwendung vorindizierter Adressierung zurückgewinnen. Dieses Verfahren ist jedoch mühsam, wenn die Parameter selbst Adressen sind.

VERSCHIEBUNG

Manchmal muß ein Unterprogramm spezielle Eigenschaften besitzen. **Ein Unterprogramm ist verschiebbar (relocatable), wenn es überall im Speicher plaziert werden kann.** Man kann ein derartiges Programm leicht verwenden, unabhängig von der Plazierung anderer Programme oder der Anordnung des Speichers. **Ein völlig verschiebbares Programm kann keine absoluten Adressen verwenden. Alle Adressen müssen relativ zum Start des Programms sein.** Ein verschiebbarer Lader ist erforderlich, um das Programm ordnungsgemäß in den Speicher zu plazieren. Der Lader wird das Programm nach anderen Programmen starten, und die Start-Adresse oder Verschiebungs-Konstante zu allen Adressen in dem Programm addieren.

WIEDEREINTRITTS-UNTERPROGRAMM

In ein Unterprogramm kann erneut eingetreten werden (es ist "re-entrant"), wenn es durch das Unterbrechungsprogramm unterbrochen wird und in das Unterprogramm erneut eingetreten werden kann, und sowohl korrekte Ergebnisse für das unterbrechende als auch das unterbrochene Programm ergibt. Die Möglichkeit des Wieder-Eintretens ist wichtig für Standard-Unterprogramme in einem System, das auf Unterbrechungen basiert. Andernfalls kann die Unterbrechungs-Service-Routine die Standard-Unterprogramme nicht verwenden, ohne daß Fehler entstehen. Mikroprozessor-Unterprogramme sind leicht derartig auszubilden, da der Aufruf-Befehl den Stapel verwendet, und dieses Verfahren ergibt den Wiedereintritt automatisch. Die einzig bleibende Forderung ist die, daß das Unterprogramm die Register und den Stapel verwendet, anstatt feste Speicherplätze für zeitweiliges Speichern. Dies ist ein wenig mühsam, kann jedoch, falls nötig, ausgeführt werden.

Ein Unterprogramm ist rekursiv, wenn es sich selbst aufruft. In ein derartiges Unterprogramm muß natürlich auch wieder eingetreten werden können. Rekursive Unterprogramme sind jedoch in Mikroprozessor-Anwendungen nicht gebräuchlich.

Die meisten Programme bestehen aus einem Hauptprogramm und mehreren Unterprogrammen. Dies ist vorteilhaft, da man erprobte Routinen verwenden, sowie die anderen Unterprogramme separat auf Fehler untersuchen und testen kann. Man muß jedoch sorgfältig bei der ordnungsgemäßen Verwendung der Unterprogramme vorgehen und sich daran erinnern, welche genauen Einflüsse sie auf Register und Speicherplätze haben.

UNTERPROGRAMM-DOKUMENTATION

DOKUMENTATION VON UNTER-PROGRAMMEN

Auflistungen von Unterprogrammen müssen genügend Informationen liefern, so daß der Anwender die interne Struktur der Unterprogramme nicht prüfen müssen. Zu diesen notwendigen Spezifikationen gehören:

- Eine Beschreibung des Zweckes des Unterprogramms
- Eine Liste der Eingabe- und Ausgabe-Parameter
- Verwendete Register und Speicherplätze
- Ein Beispiel

Wenn diese Richtlinien befolgt werden, wird das Unterprogramm leicht zu verwenden sein.

BEISPIELE

Wichtige Anmerkung: Alle folgenden Beispiele reservieren einen Speicherbereich für den RAM-Stapel.

Wenn der Monitor in Ihrem Mikrocomputer einen derartigen Bereich verwendet, können Sie ihn stattdessen verwenden. Wenn Sie Ihren eigenen Stapelbereich einrichten wollen, erinnern Sie sich daran, den Stapelzeiger des Monitors aufzubewahren und zurückzuspeichern, um eine ordnungsgemäße Rückkehr am Ende Ihres Hauptprogrammes sicherzustellen.

Um den Monitor-Stapelzeiger aufzubewahren, verwenden Sie die Befehlssequenz

```
TSX
STX    TEMP
```

Um den Monitor-Stapelzeiger zurückspeichern, verwenden Sie die Sequenz

```
LDX    TEMP
TXS
```

Beachten Sie das der Stapelzeiger nür über Register X geladen oder gespeichert werden kann. Erinnern Sie sich daran, daß der 6502 immer seinen Stapel auf Seite eins des Speichers aufbewahrt, so daß die wirkliche Stapel-Adresse 01ss ist, wobei ss der Inhalt des 8-Bit-Stapelzeiger-Registers ist.

Wir haben die Adresse $01FF_{16}$ als Startpunkt für den Stapel verwendet. Sie müssen vielleicht diese Adresse durch eine geeignetere für Ihre Konfiguration ersetzen. Hierzu sollten Sie das Anwender-Handbuch Ihres Mikrocomputers nachschlagen, um die erforderlichen Änderungen zu bestimmen.

Die grundlegende Sequenz zur Initialisierung des Stapelzeigers ist daher

```
LDX    #$FF        ;PLAZIERE STAPEL AM OBEREN ENDE VON
                   ;  SEITE 1
TXS
```

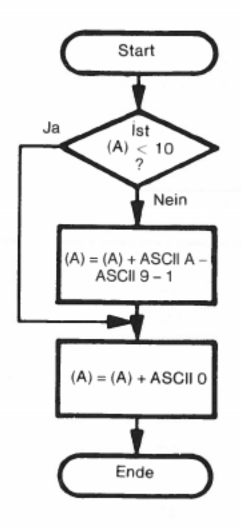

Hexadezimal in ASCII

Zweck: Wandle den Inhalt des Akkumulators von einer Hexadezimalzahl-Ziffer in ein ASCII-Zeichen um. Nimm an, daß der Akkumulator ursprünglich eine einzelne gültige hexadezimale Ziffer enthält.

Beispiele:

	(A) = 0C	
Ergebnis:	(A) = 43	ASCII C
	(A) = 06	
Ergebnis:	(A) = 36	ASCII 6

Flußdiagramm:



Das Aufruf-Programm beginnt den Stapel im Speicherplatz 01FF, holt die Daten vom Speicherplatz 0040, ruft das Umwandlungs-Unterprogramm auf und speichert das Ergebnis in den Speicherplatz 0041.

```
        *=0
        LDX   #$FF      ;PLAZIERE STAPEL AM ENDE VON SEITE 1
        TXS
        LDA   $40       ;HOLE HEXADEZIMAL-DATEN
        JSR   ASDEC     ;WANDLE DATEN IN ASCII UM
        STA   $41       ;SPEICHERE ERGEBNIS
        BRK
```

Das Unterprogramm wandelt die hexadezimalen Daten in ASCII um.

```
        *=$20
ASDEC   CMP   #10       ;SIND DIE DATEN EINE DEZIMALZIFFER?
        BCC   ASCZ
        ADC   #'A-'9-2  ;NEIN, ADDIERE OFFSET FÜR BUCHSTABEN
ASCZ    ADC   #'0       ;WANDLE IN ASCII DURCH ADDITION VON
                        ;  ASCII NULL UM
        RTS
```

Unterprogramm-Dokumentation:

```
;
;UNTERPROGRAMM ASDEC
;
;ZWECK: ASDEC WANDELT EINE HEXADEZIMAL-ZIFFER IM AKKUMULATOR
;  IN EINE ASCII-ZIFFER IM AKKUMULATOR UM
;
;ANFANGSBEDINGUNGEN: HEX-ZIFFER IN A
;
;ENDBEDINGUNGEN: ASCII-ZEICHEN IN A
;
;VERWENDETE REGISTER: A
;
;BEISPIEL
;  ANFANGSBEDINGUNGEN: 6 IM AKKUMULATOR
;  ENDBEDINGUNGEN: ASCII 6 (HEX 36)
;  IM AKKUMULATOR
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
1) Aufrufendes Programm				
0000	A2		LDX	#$FF
0001	FF			
0002	9A		TXS	
0003	A5		LDA	$40
0004	40			
0005	20		JSR	ASDEC
0006	20			
0007	00			
0008	85		STA	$41
0009	41			
000A	00		BRK	
2) Unterprogramm				
0020	C9	ASDEC	CMP	#10
0021	0A			
0022	90		BCC	ASCZ
0023	02			
0024	69		ADC	#'A-'9-2
0025	06			
0026	69	ASCZ	ADC	#'0
0027	30			
0028	60		RTS	

Die Befehle LDX #$FF und TXS beginnen den Stapel im Speicherplatz 01FF. Erinnern Sie sich daran, daß der Stapel nach unten wächst (zu niedrigeren Adressen), und daß der Stapelzeiger des 6502 immer die Adresse auf Seite 1 der nächsten leeren Speicherstelle enthält (anstatt die zuletzt gefüllte, wie bei einigen anderen Mikroprozessoren).

Der Befehl "Springe zu Unterprogramm" plaziert die Start-Adresse des Unterprogramms (0020) in den Befehlszähler und bewahrt den momentanen Stand des Befehlszählers (die Adresse des letzten Bytes des JSR-Befehls) im Stapel auf. Der Vorgang sieht folgendermaßen aus:

Schritt 1) Rette MSBs des alten Befehlszähler-Standes in den Stapelzeiger, dekrementiere Stapelzeiger.

Schritt 2) Rette LSBs des alten Befehlszähler-Standes in den Stapel, dekrementiere Stapelzeiger.

Beachten Sie, daß der Stapelzeiger dekrementiert wird, nachdem die Daten gespeichert wurden.

Die MSBs des Befehlszählers werden zuerst gespeichert, jedoch diese Bits enden bei den höheren Adressen (in der gebräuchlichen Art und Weise des 6502), da der Stapel nach abwärts wächst.

Das Ergebnis in diesem Beispiel ist:

(01FF) = 00
(01FE) = 07
(S) = FD

Der Wert, den der Befehl "Jump to Subroutine" aufbewahrt, ist der Befehlszähler, bevor das lezte Byte des JSR-Befehls geholt worden ist. Dieser Wert ist daher um 1 kleiner als die richtige Rückkehr-Adresse. Der Befehl "Return-from-Subroutine" (RTS) holt die beiden obersten Eingaben vom Stapel, addiert eins (infolge der eben erwähnten ungeraden Versetzung des 6502), und plaziert das Ergebnis zurück in den Befehlszähler. Das Verfahren lautet:

Schritt 1) Inkrementiere Stapelzeiger, lade acht Bits vom Stapel, plaziere Ergebnis in LSBs des Befehlszählers.

Schritt 2) Inkrementiere Stapelzeiger, lade acht Bits vom Stapel, plaziere Ergebnis in MSBs des Befehlszählers.

Schritt 3) Inkrementiere Befehlszähler, bevor ein Befehl wirklich geholt wird.

Hier wird der Stapelzeiger inkrementiert, bevor die Daten geladen werden.

Das Ergebnis in diesem Fall ist:

PC = (00FF)(00FE) + 1
= 0008
(SP) = FF

Dieses Unterprogramm besitzt einen einzelnen Parameter und erzeugt ein einziges Ergebnis. Ein Akkumulator ist der naheliegende Platz, in dem beide zu plazieren sind.

Das Aufruf-Programm besteht aus drei Schritten: Plazieren der Daten in den Akkumulator, Aufrufen des Unterprogrammes und Speichern des Ergebnisses in den Speicher. Die gesamte Initialisierung muß hierbei auch den Stapel in den entsprechenden Speicherbereich plazieren.

In das Unterprogramm kann wieder eingetreten werden (re-entrant), da es keinen Datenspeicher verwendet. Es ist auch verschiebbar (relocatable), da die Adresse ASCZ nur in einem Befehl für eine bedingte Verzweigung mit relativer Adressierung verwendet wird.

Beachten Sie, daß der Befehl "Springe zu Unterprogramm" in einer Ausführung von vier oder fünf Befehlen resultiert und hierbei 13 oder 14 Taktzyklen benötigt. Ein Unterprogramm-Aufruf kann daher in einer längeren Ausführungszeit resultieren, auch wenn es so aussieht, als ob er nur ein einzelner Befehl im Programm wäre.

Wenn Sie planen, den Stapel zur Übergabe von Parameter zu verwenden, so erinnern Sie sich daran, daß JSR die Rückkehr-Adresse in der Spitze des Stapels aufbewahrt.

Sie können den Stapelzeiger zum Indexregister X bringen, um auf die Daten zugreifen zu können, müssen sich jedoch daran erinnern, die richtigen Versetzungen zu liefern. Sie können auch den Zugriff auf die Daten erhalten, indem Sie zwei zusätzliche PLA-Befehle zur Übertragung des Stapelzeigers zurück zur Rückkehr-Adresse verwenden, müssen sich jedoch daran erinnern, den Stapelzeiger wieder zurück auf seinen ursprünglichen Wert vor der Rückkehr einzustellen.

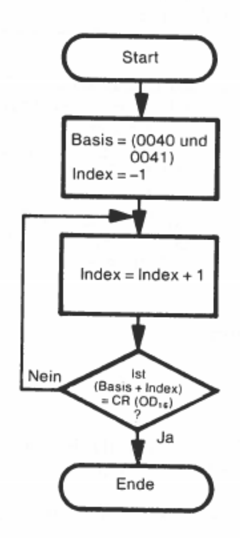
Länge einer Zeichenreihe

Zweck: Bestimme die Länge einer Reihe von ASCII-Zeichen. Die Start-Adresse der Reihe befindet sich in den Speicherplätzen 0040 und 0041. Das Ende der Reihe wird durch ein Wagenrücklauf-Zeichen (CR, $0D_{16}$) markiert. Plaziere die Länge der Reihe (ausschließlich des Wagenrücklaufs) in den Akkumulator.

Beispiele:

a.	(0040)	=	43	Startadresse der Reihe
	(0041)	=	00	
	(0043)	=	52	'R'
	(0044)	=	41	'A'
	(0045)	=	54	'T'
	(0046)	=	48	'H'
	(0047)	=	45	'E'
	(0048)	=	52	'R'
	(0049)	=	0D	'CR'
Ergebnis:	(A)	=	06	
b.	(0040)	=	0343	Startadresse der Reihe
	(0041)	=	00	
	(0043)	=	0D	
Ergebnis:	(A)	=	00	

Flußdiagramm:



Quellprogramm:

Das Aufruf-Programm startet den Stapel im Speicherplatz 01FF, speichert die Start-Adresse der Reihe in die Speicherplätze 0040 und 0041, ruft das Unterprogramm für die Reihenlänge auf und speichert das Ergebnis in den Speicherplatz 0042.

Die Speicherplätze 0040 und 0041 werden verwendet, als ob sie zusätzliche Register wären.

```
*=0
LDX   #$FF     ;PLAZIERE STAPEL AM ENDE VON SEITE 1
TXS
LDA   #$43     ;BEWAHRE STARTADRESSE DER REIHE AUF
STA   $40
LDA   #0
STA   $41
JSR   $TLEN    ;BESTIMME LÄNGE DER REIHE
STA   $42      ;SPEICHERE REIHENLÄNGE
BRK
```

Das Unterprogramm bestimmt die Länge der Reihe von ASCII-Zeichen und plaziert die Länge in den Akkumulator B.

```
        * = $20
STLEN   LDY   #$FF          ;REIHENLÄNGE = 1
        LDA   #$0D          ;HOLE ASCII-WAGENRÜCKLAUF FÜR VER-
                            ;  GLEICH
CHKCR   INY                 ;ADDIERE 1 ZUR REIHENLÄNGE
        CMP   ($40),Y       ;IST NÄCHSTES ZEICHEN EIN WAGENRÜCK-
                            ;  LAUF?
        BNE   CHKCR         ;NEIN, HALTE WEITER AUSSCHAU
        TYA                 ;BEWAHRE REIHENLÄNGE IN AKKUMULA-
                            ;  TOR AUF
        RTS
```

Unterprogramm-Dokumentation:

```
;
;UNTERPROGRAMM
;
;ZWECK: STLEN BESTIMMT DIE LÄNGE EINER ASCII-REIHE (ANZAHL DER
;  ZEICHEN VOR EINEM WAGENRÜCKLAUF)
;
;ANFANGSBEDINGUNGEN: START-ADRESSE DER REIHE IN DEN SPEICHER-
;  PLÄTZEN 0040 UND 0041
;
;ENDBEDINGUNGEN: ANZAHL DER ZEICHEN IN A
;
;VERWENDETE REGISTER: A, Y, ALLE FLAGS AUSSER ÜBERLAUF
;VERWENDETE SPEICHERPLÄTZE: 0040, 0041
;
;BEISPIEL:
;ANFANGSBEDINGUNGEN: 0043 IN DEN SPEICHERPLÄTZEN 0040 UND
;  0041
;  (0043) = 35, (0044) = 46, (0045) = 0D
;
;  ENDBEDINGUNGEN: (A) = 02
;
```


Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
1) Aufrufendes Programm				
0000	A2		LDX	#$FF
0001	FF			
0002	9A		TXS	
0003	A9		LDA	#$43
0004	43			
0005	85		STA	$40
0006	40			
0007	A9		LDA	#0
0008	00			
0009	85		STA	$41
000A	41			
000B	20		JSR	STLEN
000C	20			
000D	00			
000E	85		STA	$42
000F	42			
0010	00		BRK	
2) Unterprogramm				
0020	A0	STLEN	LDY	#$FF
0021	FF			
0022	A9		LDA	#$0D
0023	0D			
0024	C8	CHKCR	INY	
0025	D1		CMP	($40),Y
0026	40			
0027	D0		BNE	CHKCR
0028	FB			
0029	98		TYA	
002A	60		RTS	

Das aufrufende Programm besteht aus vier Schritten: Initialisieren des Stapelzeigers, Plazieren der Startadresse der Reihen in die Speicherplätze 0040 und 0041, Aufrufen des Unterprogramms und Speichern des Ergebnisses.

In das Unterprogramm kann nicht wieder eingetreten werden, da es feste Speicheradressen 0040 und 0041 verwendet. Wenn diese Speicherplätze jedoch als zusätzliche Register betrachtet werden und ihr Inhalt automatisch aufbewahrt und mit den Anwender-Registern zurückgespeichert wird, kann das Unterprogramm auf eine "Wiedereintrittsart" verwendet werden. Zahlreiche Computer aller Größen verwenden Register, die in Wirklichkeit im Speicher liegen. Diese Lösung macht das Speicher-Management komplexer, ändert jedoch nicht die grundlegenden Verfahren.

Das Unterprogramm ändert das Register Y, sowie den Inhalt des Akkumulators. Der Programmierer muß daher darauf achten, daß die im Index-Register Y gespeicherten Daten verloren gehen. Die Dokumentation des Unterprogramms muß beschreiben, wozu die Register verwendet werden.

Ein Weg zum Aufbewahren des Register-Inhaltes während eines Unterprogrammes besteht darin, sie in den Stapel zu legen und dann vor der Rückkehr zurückzuspeichern. Diese Lösung erleichtert das Leben für den Anwender der Routine, kostet jedoch zusätzliche Zeit und Speicher (im Programm und im Stapel). Um das Indexregister Y zu retten und zurückzuspeichern, müßten Sie folgende Sequenz hinzufügen

```
TYA         ;BEWAHRE ALTEN INHALT VON Y AUF
PHA
```

Zu Beginn des Programmes und

```
PLA         ;SPEICHERE ALTEN INHALT VON Y ZURÜCK
TAY
```

am Ende des Programmes.

Dieses Unterprogramm besitzt einen einzelnen Eingangs-Parameter, der eine Adresse ist. Der einfachste Weg um diesen Parameter zu übertragen, geschieht über zwei Speicherplätze auf der Nullseite. Der 6502 besitzt keine Register mit voller Adressenlänge, mit denen diese Parameter übertragen werden könnten.

Wenn das Abschlußzeichen nicht immer ein ASCII-Wagenrücklauf wäre, könnten wir dieses Zeichen in einen anderen Parameter umwandeln. Nun müßte das aufrufende Programm das Abschlußzeichen in den Akkumulator plazieren und die Start-Adresse der Reihe in die Speicherplätze 0040 und 0041, bevor das Unterprogramm aufgerufen wird.

Ein Weg zum Übertragen von Parametern, die für einen bestimmten Aufruf festliegen, besteht im Plazieren ihrer Werte in den Programmspeicher, unmittelbar nach dem Befehl Jump-to-Subroutine.[1] Sie können den alten Befehlszähler (aufbewahrt in der Spitze des Stapels) zum Zugriff auf die Daten verwenden, müssen jedoch die Rückkehr-Adresse einrichten (Erhöhung um die Anzahl der für Parameter verwendeten Bytes), bevor die Steuerung zurück zum Hauptprogramm transferiert wird. Beispielsweise könnten wir den Wert des Abschlußzeichens auf diese Weise übertragen. Das Hauptprogramm würde die Pseudo-Operation .BYTE'. unmittelbar nach dem JSR-Befehl enthalten. Das Unterprogramm könnte die Rückkehr-Adresse in die Speicherplätze 0050 und 0051 plazieren und auf die verschiedenen Parameter unter Verwendung von Nach-Indizierung zugreifen. Die folgende Sequenz könnte die Rückkehr-Adresse aufbewahren, wobei man sich erinnere, daß der Stapel immer auf Seite 1 des Speichers liegt und der Stapelzeiger immer die Adresse der nächsten verfügbaren Speicherstelle enthält.

```
TSX                  ;HOLE STPELZEIGER
LDA    $0101,X       ;HOLE MSB'S DER RÜCKKEHRADRESSE
STA    $50
LDA    $0102,X       ;HOLE LSB'S DER RÜCKKEHRADRESSE
STA    $51
```

Achten Sie sehr sorgfältig auf die Tatsache, daß die Rückkehr-Adresse in Wirklichkeit die Adresse des letzten (dritten) Bytes des JSR-Befehls ist, nicht die Adresse unmittelbar nach dem JSR-Befehl, wie dies bei den meisten anderen Mikroprozessoren der Fall ist. Die tatsaächliche Rückkehr-Adresse muß daher um 1 versetzt werden, da RTS automatisch 1 zu ihr addieren wird.

Die Befehle PHA (Store Accumulator in Stack) und PLA (Load Accumulator from Stack) transferieren acht Datenbits zwischen dem Akkumulator und dem RAM-Stapel. Die Indexregister X und Y können nur über den Akkumulator aufbewahrt und zurückgespeichert werden. Wie beim Befehl Jump-to-Subroutine wird der Stapelzeiger dekrementiert, nachdem die Daten in den Stapel gespeichert wurden und inkrementiert, bevor die Daten von ihm geladen werden. Erinnern Sie sich daran, daß der RAM-Stapel abwärts wächst (zu niedrigeren Adressen).

Maximalwert

Zweck: Suche das größte Element in einem Block von Binärzahlen ohne Vorzeichen. Die Länge des Blockes befindet sich im Indexregister Y und die Startadresse des Blocks liegt in den Speicherplätzen 0040 und 0041. Der Maximalwert wird in den Akkumulator zurückgelegt.

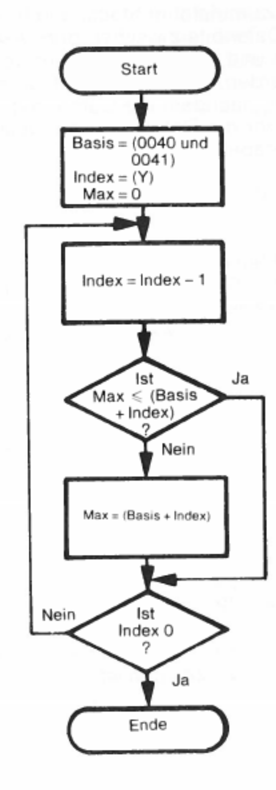
Beispiele:

```
    (Y) = 05  Länge des Blocks
 (0040) = 43  Startadresse des Blocks
 (0041) = 00

 (0043) = 67
 (0044) = 79
 (0045) = 15
 (0046) = E3
 (0047) = 72
```

Ergebnis: (A) = E3, da dies die größte von fünf Zahlen ohne Vorzeichen ist.

Flußdiagramm:



Quellprogramm:

Das aufrufende Programm startet den Stapel im Speicherplatz 01FF, setzt die Start-Adresse des Blockes auf 0043, holt die Blocklänge vom Speicherplatz 0030, ruft das Unterprogramm für den Maximalwert auf und speichert das Maximum in den Speicherplatz 0042.

```
        *=0
        LDX     #$FF        ;PLAZIER STAPEL AN DAS ENDE DER
                            ;  SEITE 1
        TXS
        LDA     #$43        ;BEWAHRE STARTADRESSE DES BLOCKS
                            ;  AUF
        STA     $40
        LDA     #0
        STA     $41
        LDY     $30         ;HOLE LÄNGE DES  BLOCKES
        JSR     MAXM        ;SUCHE MAXIMALWERT
        STA     $42         ;BEWAHRE MAXIMALWERT AUF
        BRK
```


Das Unterprogramm bestimmt den Maximalwert in dem Block

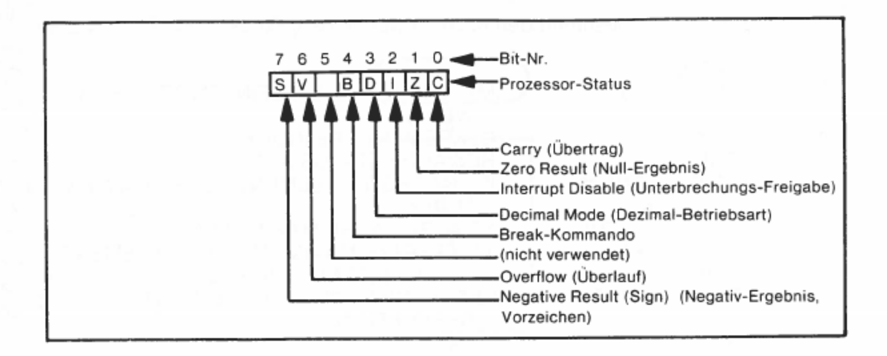
```
        *=S20
MAXM    LDA     #0          ;MAXIMUM = NULL (KLEINSTMÖGLICHER
                            ;  WERT)
CMPE    DEY                 ;DEKREMENTIERE INDEX
        PHP                 ;BEWAHRE STATUS AUF
        CMP     ($40),Y     ;IST NÄCHSTES ELEMENT ÜBER DEM MAXI-
                            ;  MUM?
        BCS     NOCHG       ;NEIN, BEWAHRE MAXIMUM AUF
        LDA     ($40),Y     ;JA, ERSETZE MAXIMUM DURCH ELEMENT
NOCHG   PLP                 ;SPEICHERE STATUS ZURÜCK
        BNE     CMPE        ;SETZE FORT,  BIS ALLE ELEMENTE
                            ;  GEPRÜFT SIND
        RTS
```

Unterprogramm-Dokumentation:

```
;
;UNTERPROGRAMM MAXM
;
;ZWECK: MAXM BESTIMMT DEN MAXIMALWERT IN EINEM BLOCK VON
;  BINÄRZAHLEN OHNE VORZEICHEN
;
;ANFANGSBEDINGUNGEN: STARTADRESSE DES BLOCKS IN DEN SPEICHER-
;  PLÄTZEN 0040 UND 0041, LÄNGE DES BLOCKS IN Y
;
;ENDBEDINGUNGEN:  MAXIMALWERT IN A
;
;VERWENDETE REGISTER: A, Y, ALLE FLAGS MIT AUSNAHME VON  ÜBER-
;  LAUF
;
;VERWENDETE SPEICHERPLÄTZE: 0040, 0041
;
;BEISPIEL:
;  ANFANGSBEDINGUNGEN: 0043 IN DEN SPEICHERPLÄTZEN 0040 UND
;    0041
;  (Y) = 03, (0043) = 35, (0040) = 46, (0045) = 0D
;  ENDBEDINGUNGEN: (A) = 46
```

Dieses Unterprogramm hat zwei Parameter – eine Adresse und eine Zahl. Die Speicherplätze 0040 und 0041 werden zur Übergabe der Adresse verwendet, und das Indexregister Y dient zur Übergabe der Zahl. Das Ergebnis ist eine einzelne Zahl, die in den Akkumulator zurückgelegt wird.

Das aufrufende Programm muß die Start-Adresse des Blockes in die Speicherplätze 0040 und 0041 plazieren und die Länge des Blockes in das Indexregister Y, bevor die Steuerung zum Unterprogramm transferiert wird.



Das Unterprogramm gibt bei null die Steuerung in das Indexregister Y zurück. Es kann in dieses nicht wieder eingetreten werden, außer die Speicherplätze 0040 und 0041 werden als zusätzliche Register behandelt. Es ist verschiebbar, da die Adressen relativ sind und der Stapel für eine zeitweilige Speicherung verwendet wird.

Beachten Sie die Verwendung der Befehle PHP und PLP, die die Statusregister aufbewahren und zurückspeichern. Dieses Register ist wie in Bild 10-1 organisiert. Wir könnten das Programm organisieren und die Anfangsbedingungen ändern, um den Bedarf für diese Befehle (siehe Kapitel 5) zu eliminieren. Der Schlüssel liegt hierbei darin, die Adresse um eins vor dem Start der Anordnung als Parameter zu liefern. Dies kann sehr leicht mit den meisten Assemblern geschehen, da sie einfache arithmetische Ausdrücke (wie etwa START–1) im Operandenfeld gestatten (siehe Kapitel 3). Der Anwender des Unterprogrammes muß jedoch gewarnt werden, daß diese Versetzung erforderlich ist.

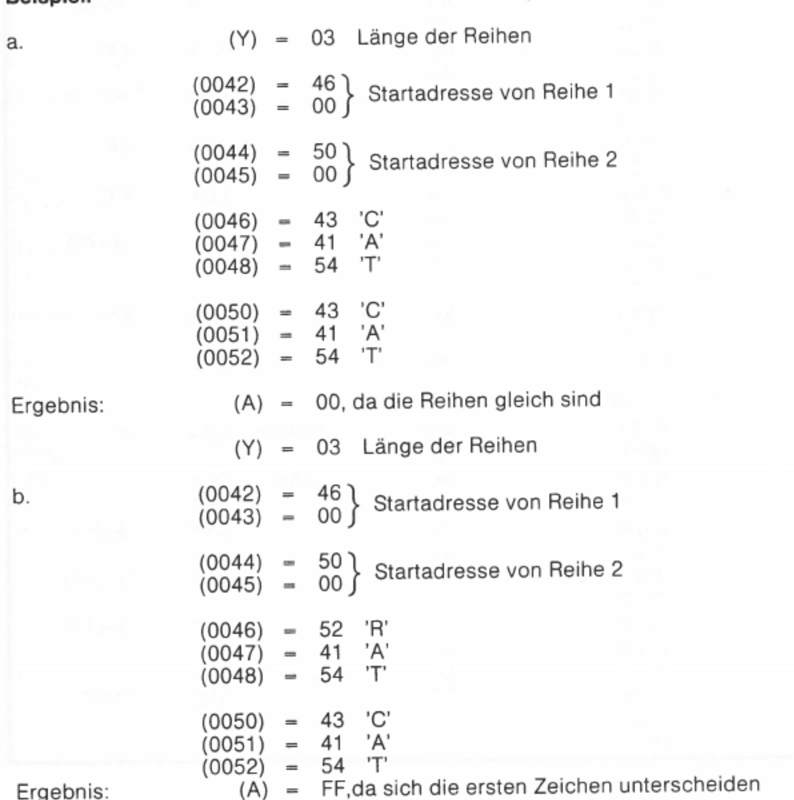
Objektprogramm

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
1) Aufrufendes Programm				
0000	A2		LDX	#$FF
0001	FF			
0002	9A		TXS	
0003	A9		LDA	#$43
0004	43			
0005	85		STA	$40
0006	40			
0007	A9		LDA	#0
0008	00			
0009	85		STA	$41
000A	41			
000B	A4		LDY	$30
000C	30			
000D	20		JSR	MAXM
000E	20			
000F	00			
0010	85		STA	$42
0011	42			
0012	00		BRK	
2) Unterprogramm				
0020	A9	MAXM	LDA	#0
0021	00			
0022	88	CMPE	DEY	
0023	08		PHP	
0024	D1		CMP	($40),Y
0025	40			
0026	B0		BCS	NOCHG
0027	02			
0028	B1		LDA	($40),Y
0029	40			
002A	28	NOCHG	PLP	
002B	D0		BNE	CMPE
002C	F5			
002D	60		RTS	

Übereinstimmung von Mustern[2]

Zweck: Vergleiche zwei Reihen von ASCII-Zeichen, um festzustellen, ob sie gleich sind. Die Länge der Reihen befindet sich im Index-Register Y. Die Start-Adresse einer Reihe liegt in den Speicherplätzen 0042 und 0043. Die Start-Adresse der anderen liegt in den Speicherplätzen 0044 und 0045. Wenn die beiden Reihen übereinstimmen, lösche Akkumulator. Andernfalls setze Akkumulator auf FF_{16}.

Beispiel:



a.

(Y) = 03 Länge der Reihen

(0042) = 46 }
(0043) = 00 } Startadresse von Reihe 1

(0044) = 50 }
(0045) = 00 } Startadresse von Reihe 2

(0046) = 43 'C'
(0047) = 41 'A'
(0048) = 54 'T'

(0050) = 43 'C'
(0051) = 41 'A'
(0052) = 54 'T'

Ergebnis: (A) = 00, da die Reihen gleich sind

(Y) = 03 Länge der Reihen

b.

(0042) = 46 }
(0043) = 00 } Startadresse von Reihe 1

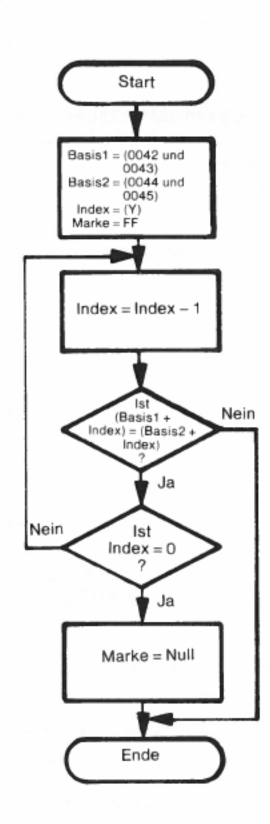
(0044) = 50 }
(0045) = 00 } Startadresse von Reihe 2

(0046) = 52 'R'
(0047) = 41 'A'
(0048) = 54 'T'

(0050) = 43 'C'
(0051) = 41 'A'
(0052) = 54 'T'

Ergebnis: (A) = FF,da sich die ersten Zeichen unterscheiden

Flußdiagramm:



Quellprogramm:

Das aufrufende Programm beginnt den Stapel im Speicherplatz 01FF, setzt die beiden Start-Adressen auf 0046 und 0050, holt die Reihenlänge vom Speicherplatz 0041, ruft das Unterprogramm für die Musterübereinstimmung auf und plaziert das Ergebnis in den Speicherplatz 0040.

```
        *=0
        LDX  #$FF         ;PLAZIERE STAPEL AM ENDE VON SEITE 1
        TXS
        LDA  #$46         ;BEWAHRE STARTADRESSE VON REIHE 1
                          ;  AUF
        STA  $42
        LDA  #0
        STA  $43
        LDA  #$50         ;BEWAHRE STARTADRESSE VON REIHE 2
                          ;  AUF
        STA  $44
        LDA  #0
        STA  $45
        LDY  $41          ;HOLE REIHENLÄNGE
        JSR  PMTCH        ;PRÜFE AUF ÜBEREINSTIMMUNG
        STA  $40          ;BEWAHRE ÜBEREINSTIMMUNGS-
                          ;  INDIKATOR AUF
        BRK
```

Das Unterprogramm bestimmt, ob die beiden Reihen übereinstimmen.

```
        *=$20
PMTCH   LDX  #$FF         ;MARKE = FF (HEX) FÜR KEINE ÜBEREIN-
                          ;  STIMMUNG
CMPE    DEY
        LDA  ($42),Y      ;HOLE ZEICHEN VON REIHE 1
        CMP  ($44),Y      ;GIBT ES EINE ÜBEREINSTIMMUNG MIT
                          ;  REIHE 2?
        BNE  DONE         ;NEIN, DONE - REIHEN STIMMEN NICHT
                          ;  ÜBEREIN
        TYA               ;SPEICHERE STATUS VON INDEX ZURÜCK
        BNE  CMPE
        LDX  #0           ;MARKE = NULL, REIHEN STIMMEN ÜBEREIN
DONE    TXA
        RTS
```


Unterprogramm-Dokumentation:

```
;
;UNTERPROGRAMM PMTCH (PATTERN MATCH)
;
;ZWECK: PMTCH BESTIMMT, OB ZWEI REIHEN ÜBEREINSTIMMEN
;
;ANFANGSBEDINGUNGEN: STARTADRESSE DER REIHEN IN DEN SPEICHER-
;  PLÄTZEN 0042 UND 0043, 0044 UND 0045. LÄNGE DER REIHEN IM INDEX-
;  REGISTER Y
;
;ENDBEDINGUNGEN: NULL IN A, WENN REIHEN ÜBEREINSTIMMEN,
;  ANDERNFALLS FF IN A
;
;VERWENDETE REGISTER: A, X, Y, ALLE FLAGS AUSSER ÜBERLAUF
;
;VERWENDETE SPEICHERPLÄTZE: 0042, 0043, 0044, 0045
;
;BEISPIEL:
;
;  ANFANGSBEDINGUNGEN: 0046 IN 0042 UND 0043, 0050 IN 0044 UND
;    0045, (Y) = 02
;    (0046) = 36, (0047) = 39,
;    (0056) = 36, (0051) = 39
;
;  ENDBEDINGUNGEN: (A) = 0, DA DIE REIHEN ÜBEREINSTIMMEN
;
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
1) Aufrufendes Programm				
0000	A2		LDX	#$FF
0001	FF			
0002	9A		TXS	
0003	A9		LDA	#$46
0004	46			
0005	85		STA	$42
0006	42			
0007	A9		LDA	#0
0008	00			
0009	85		STA	$43
000A	43			
000B	A9		LDA	#$50
000C	50			
000D	85		STA	$44
000E	44			
000F	A9		LDA	#0
0010	00			
0011	85		STA	$45
0012	45			
0013	A4		LDY	$41
0014	41			
0015	20		JSR	PMTCH
0016	20			
0017	00			
0018	85		STA	$40
0019	40			
001A	00		BRK	
2) Unterprogramm				
0020	A2	PMTCH	LDX	#$FF
0021	FF			
0022	88	CMPE	DEY	
0023	B1		LDA	($42),Y
0024	42			
0025	D1		CMP	($44),Y
0026	44			
0027	D0		BNE	DONE
0028	05			
0029	98		TYA	
002A	D0		BNE	CMPE
002B	F6			
002C	A2		LDX	#0
002D	00			
002E	8A	DONE	TXA	
002F	60		RTS	

Dieses Unterprogramm ändert, wie das vorausgehende, alle Flags mit Ausnahme des Überlaufs. Sie sollten im allgemeinen annehmen, daß ein Unterprogramm-Aufruf die Flags ändert, außer es ist andernfalls speziell festgelegt. Wenn das Hauptprogramm die alten Flag-Werte benötigt (für spätere Prüfung) muß sie diese in den Stapel vor dem Aufruf des Unterprogrammes aufbewahren. Dies geschieht mit dem PHP-Befehl.

Dieses Unterprogramm verwendet alle Register und vier Speicherplätze auf Seite null. Es gibt drei Parameter - zwei Start-Adressen und die Länge der Reihe.

Der Befehl TYA hat keinen anderen Zweck, als das Null-Flag entsprechend dem Inhalt von Indexregister Y zu setzen. Wir könnten den Bedarf für diesen Befehl eliminieren, indem wir das Unterprogramm neu organisieren. Eine Alternative wäre die Änderung der Parameter, so daß die Adressen beide um 1 versetzt wären (d.h., beide Reihen-Adressen würden sich in Wirklichkeit auf das Byte beziehen, das unmittelbar vor der Zeichenreihe kommt). Erinnern Sie sich jedoch daran, daß der Anwender in der Lage sein sollte, Parameter in das Unterprogramm auf die einfachste und möglichst offensichtlichste Weise zu übertragen. Der Anwender sollte nicht Adressen um 1 versetzen oder andere Anordnungen für das Unterprogramm ausführen müssen. Derartigen Praktiken resultieren in zahlreichen unangenehmen Programmierfehlern. Das Programm sollte derartige Dinge nur ausführen, wenn Zeit oder Speicher-Überlegungen kritisch sind.

Eine weitere Alternative bestünde im Dekrementieren des Index um 1 zu Beginn, um das Problem des Zugriffes nach dem Ende der Reihe zu vermeiden. Das Ende der Schleife würde dann den Index dekrementieren und zurück verzweigen, solange das Resultat positiv ist, das heißt

```
DEY
BPL     CMPE
```

Diese Lösung würde solange arbeiten, solange die Reihe kürzer als 130 Bytes ist. Die Begrenzung tritt auf, da das Vorzeichen-Flag des 6502 gesetzt wird, wenn das Ergebnis einer Zahl ohne Vorzeichen größer als 127 (dezimal) ist.

Addition mit mehrfacher Genauigkeit

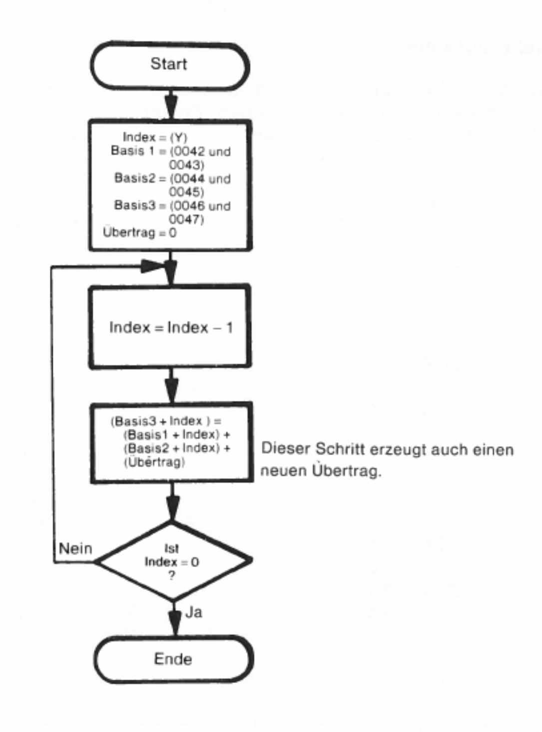
Zweck: Addiere zwei Mehrwort-Binärzahlen. Die Länge der Zahlen (in Bytes) befindet sich im Index-Register Y, die Start-Adressen der Zahlen in den Speicherplätzen 0042 und 0043, und in 0044 und 0045, und die Start-Adresse des Ergebnisses befindet sich in den Speicherplätzen 0046 und 0047. Alle Zahlen beginnen mit den höchstwertigen Bits.

Beispiel:

	Adresse		Wert	Bedeutung
	(Y)	=	04	Länge der Zahlen in Bytes
	(0042)	=	48	Startadresse der ersten Zahl
	(0043)	=	00	
	(0044)	=	4C	Startadresse der zweiten Zahl
	(0045)	=	00	
	(0046)	=	50	Startadresse des Ergebnisses
	(0047)	=	00	
	(0048)	=	2F	MSBs der ersten Zahl
	(0049)	=	5B	
	(004A)	=	A7	
	(004B)	=	C3	LSBs der ersten Zahl
	(004C)	=	14	MSBs der zweiten Zahl
	(004D)	=	DF	
	(004E)	=	35	
	(004F)	=	B8	LSBs der zweiten Zahl
Ergebnis:	(0050)	=	44	MSBs des Ergebnisses
	(0051)	=	3A	
	(0052)	=	DD	
	(0053)	=	7B	LSBs des Ergebnisses

d.h.
```
  2F5BA7C3
+ 14DF35B8
  443ADD7B
```


Flußdiagramm:



Quellprogramm:

Das Aufruf-Programm beginnt den Stapel im Speicherplatz 01FF, setzt die Start-Adressen der verschiedenen Zahlen entsprechend auf 0048, 004C und 0050, holt die Länge der Zahlen vom Speicherplatz 0040 und ruft das Unterprogramm für die Addition mit mehrfacher Genauigkeit auf.

```
        *=0
        LDX   #$FF        ;PLAZIERE STAPEL AN DAS ENDE VON
                          ;  SEITE 1
        TXS
        LDA   #$48        ;BEWAHRE STARTADRESSE DER ERSTEN
                          ;  ZAHL AUF
        STA   $42
        LDA   #$4C        ;BEWAHRE STARTADRESSE DER ZWEITEN
                          ;  ZAHL AUF
        STA   $44
        LDA   #$50        ;BEWAHRE STARTADRESSE DES
                          ;  ERGEBNISSES AUF
        STA   $46
        LDA   #0          ;BEWAHRE SEITENNUMMER FÜR ALLE
                          ;  ADRESSEN AUF
        STA   $43
        STA   $45
        STA   $47
        LDY   $40         ;HOLE LÄNGE DER ZAHLEN IN BYTES
        JSR   MPADD       ;ADDITION MIT MEHRFACHER GENAUIG-
                          ;  KEIT
        BRK
```

Das Unterprogramm führt die Binär-Addition mit mehrfacher Genauigkeit aus.

```
        *=$20
MPADD   CLC               ;LÖSCHE ÜBERTRAG ZU BEGINN
ADDB    DEY
        LDA   ($42),Y     ;HOLE BYTE VON ERSTER ZAHL
        ADC   ($44),Y     ;ADDIERE BYTE VON ZWEITER ZAHL
        STA   ($46),Y     ;SPEICHERE ERGEBNIS
        TYA               ;ALLE BYTES ADDIERT?
        BNE   ADDB        ;NEIN, SETZE FORT
        RTS
```


Unterprogramm-Dokumentation:

```
;UNTERPROGRAMM MPADD (MULTI-PRECISION ADDITION)
;
;ZWECK: MPADD ADDIERT ZWEI MEHRWORT-BINÄRZAHLEN
;
;ANFANGSBEDINGUNGEN: STARTADRESSEN DER ZAHLEN IN DEN
;  SPEICHERPLÄTZEN 0042 UND 0043, 0044 UND 0045.
;  STARTADRESSE DES ERGEBNISSES IN DEN SPEICHERPLÄTZEN 0046 UND
;  0047.
;  LÄNGE DER ZAHLEN IM INDEX-REGISTER Y
;
;VERWENDETE REGISTER: A, Y, ALLE FLAGS
;
;VERWENDET SPEICHERPLÄTZE: 0042, 0043, 0044, 0045, 0046, 0047,
;
;BEISPIEL:
;  ANFANGSBEDINGUNGEN: 0048 IN 0042 UND 0043,
;    004C IN 0044 UND 0045, 0050 IN 0046 UND 0047,
;    (Y)=02, (0048)=A7, (0049)=Ç C3, (0046)=35, (004D)=B8
;
;  ENDBEDINGUNGEN: (0050)=DD, (0051)=7B
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
1) Aufrufendes Programm				
0000	A2		LDX	#$FF
0001	FF			
0002	9A		TXS	
0003	A9		LDA	#$48
0004	48			
0005	85		STA	$42
0006	42			
0007	A9		LDA	#$4C
0008	4C			
0009	85		STA	$44
000A	44			
000B	A9		LDA	#$50
000C	50			
000D	85		STA	$46
000E	46			
000F	A9		LDA	#0
0010	00			
0011	85		STA	$43
0012	43			
0013	85		STA	$45
0014	45			
0015	85		STA	$47
0016	47			
0017	A4		LDY	$40
0018	40			
0019	20		JSR	MPADD
001A	20			
001B	00			
001C	00		BRK	
2) Unterprogramm				
0020	18	MPADD	CLC	
0021	88	ADDB	DEY	
0022	B1		LDA	($42),Y
0023	42			
0024	71		ADC	($44),Y
0025	44			
0026	91		STA	($46),Y
0027	46			
0028	98		TYA	
0029	D0		BNE	ADDB
002A	F6			
002B	60		RTS	

Dieses Unterprogramm besitzt vier Parameter - drei Adressen und die Länge der Zahlen. Für die Übergabe der Parameter werden sechs Speicherplätze auf Seite null und das Indexregister Y verwendet.
Wie beim vorhergehenden Beispiel könnten wir den TYA-Befehl durch Neuorganisieren des Programmes eliminieren, oder durch Versetzen der Adressen-Parameter um 1.

AUFGABEN

Beachten Sie, daß Sie sowohl das Aufruf-Programm für das Beispiel, als auch ein entsprechend dokumentiertes Unterprogramm schreiben sollen.

1) ASCII in Hexadezimal

Zweck: Wandle den Inhalt des Akkumulators von der ASCII-Darstellung einer Hexadezimal-Ziffer in die tatsächliche Ziffer um. Plaziere das Ergebnis in den Akkumulator.

Beispiele:

a. (A) = 43 ASCII C

Ergebnis: (A) = 0C

b. (A) = 36 ASCII 6

Ergebnis: (A) = 06

2) Länge einer Fernschreib-Nachricht

Zweck: Bestimme die Länge einer ASCII-codierten Fernschreib-Nachricht. Die Start-Adresse der Zeichen-Reihe, in der die Nachricht eingebettet ist, liegt in den Speicherplätzen 0043 und 0043. Die Nachricht selbst beginnt mit einem ASCII-Zeichen STX (02_{16}) und endet mit ASCII-ETX (03_{16}). Plaziere die Länge der Nachricht (Anzahl der Zeichen zwischen dem STX und dem ETX) in den Akkumulator.

Beispiel:

(0042) = 44 } Startadresse der Reihe
(0043) = 00 }

(0044) = 49
(0045) = 02 STX
(0046) = 47 'G'
(0047) = 4F 'O'
(0048) = 03 ETX

Ergebnis: (A) = 02

3) Minimalwert

Zweck: Suche das kleinste Element in einem Block von Binärzahlen ohne Vorzeichen. Die Länge des Blocks liegt im Index-Register Y und die Startadresse des Blockes liegt in den Speicherplätzen 0040 und 0041. Der Minimalwert wird in den Akkumulator zurückgelegt.

Beispiel:

(Y) = 05 Länge des Blockes

(0040) = 43 } Startadresse des Blockes
(0041) = 00 }

(0043) = 67
(0044) = 79
(0045) = 15
(0046) = E3
(0047) = 73

Ergebnis: (A) = 15, da dies die kleinste von fünf Zahlen ohne Vorzeichen ist.

4) Vergleich von Reihen

Zweck: Vergleiche zwei Reihen von ASCII-Zeichen um festzustellen, welche größer ist (d.h. welche der anderen in "alphabetischer" Reihenfolge folgt). Die Länge der Reihen befindet sich im Index-Register Y, die Start-Adresse der Reihe 1 in den Speicherplätzen 0042 und 0043, und die Start-Adresse der Reihe 2 in den Speicherplätzen 0044 und 0045. Wenn Reihe 1 größer oder gleich wie Reihe 2 ist, lösche Akkumulator. Andernfalls setze Akkumulator auf FF_{16}.

Beispiele:

a. (Y) = 03 Länge der Reihen

(0042) = 46 }
(0043) = 00 } Startadresse von Reihe 1

(0044) = 4A }
(0045) = 00 } Startadresse der Reihe 2

(0046) = 43 'C'
(0047) = 42 'A'
(0048) = 54 'T'

(004A) = 42 'B'
(004B) = 41 'A'
(004C) = 54 'T'

Ergebnis: (A) = 00, da 'CAT' "größer als 'BAT' ist

b. (Y) = 03 Länge der Reihen

(0042) = 46 }
(0043) = 00 } Startadresse der Reihe 1

(0044) = 4A }
(0045) = 00 } Startadresse der Reihe 2

(0046) = 43 'C'
(0047) = 41 'A'
(0048) = 54 'T'

(004A) = 43 'C'
(004B) = 41 'A'
(004C) = 54 'T'

Ergebnis: (A) = 00, da die Reihen gleich sind

c. (Y) = 03 Länge der Reihen

(0042) = 46 }
(0043) = 00 } Startadresse der Reihe 1

(0044) = 4A }
(0045) = 00 } Startadresse der Reihe 2

(0046) = 43 'C'
(0047) = 41 'A'
(0048) = 54 'T'

(004A) = 43 'C'
(004B) = 55 'U'
(004C) = 54 'T'

Ergebnis: (A) = FF, da 'CUT' "größer" als 'CAT' ist.

5) Dezimale Subtraktion

Zweck: Subtrahiere eine Mehrwort-Dezimalzahl (BCD) von einer anderen. Die Länge der Zahlen (in Bytes) befindet sich im Index-Register Y und die Start-Adressesn der Zahlen in den Speicherplätzen 0042 und 0043, 0044 und 0045. Subtrahiere die Zahl mit der Start-Adresse in 0044 und 0045 vor der mit der Start-Adresse in 0042 und 0043. Die Start-Adresse des Ergebnisses befindet sich in den Speicherplätzen 0046 und 0047. Alle Zahlen beginnen mit den höchstwertigen Ziffern. Das Vorzeichen des Ergebnisses wird in den Akkumulator zurückgelegt - null, wenn das Resultat positiv ist, FF_{16}, wenn es negativ ist.

Beispiel:

(Y) = 04 Länge der Zahlen in Bytes

(0042) = 48 }
(0043) = 00 } Startadresse des Minuenden

(0044) = 4C }
(0045) = 00 } Startadresse des Subtrahenden

(0046) = 50 }
(0047) = 00 } Startadresse der Differenz

(0048) = 36 höchstwertige Stelle des Minuenden
(0049) = 70
(004A) = 19
(004B) = 85 niedrigstwertige Stelle des Minuenden

(004C) = 12 höchstwertige Stelle des Subtrahenden
(004D) = 66
(004E) = 34
(004F) = 59 niedrigstwertige Stelle des Subtrahenden

Ergebnis: (A) = 00 positives Ergebnis

(0050) = 24 höchstwertige Stelle der Differenz
(0051) = 03
(0052) = 85
(0053) = 26 niedrigstwertige Stelle der Differenz

d.h. 36701985
− 12663459
+ 24038526

LITERATUR

1) Andere Beispiele für dieses Verfahren (für den 8080) sind enthalten in: S. Mazor and C. Pitchford, "Develop Cooperative Microprocessor Subroutines," Electronic Design, June 7, 1978, pp. 116-118.

2) J. T. O Donnell, "6502 Routine Compares Character Strings," EDN, August 5, 1978, p. 54.

Kapitel 11
EINGABE/AUSGABE

Es gibt zwei Probleme bei der Entwicklung von Eingabe/Ausgabe Anordnungen: Ein Problem besteht darin, wie periphere Bausteine an den Computer anzupassen sind und wie Daten, Status und Stuersignale übertragen können. Das andere besteht im Adressieren der E/A-Bausteine, so daß die CPU einen speziellen hiervon für einen Datentransfer auswählen kann. Das erste Problem ist offensichtlich sowohl komplexer als auch interessanter. **Wir wollen daher hier das Anpassen (Interfacing) von peripheren Geräten besprechen und die Adressierung einem mehr hardware-orientierten Buch überlassen.**

E/A UND SPEICHER

Theoretisch ist der Transfer von Daten von oder zu einem E/A-Baustein ähnlich dem Transfer von Daten zu oder von einem Speicher. In der Tat können wir den Speicher wie jeden beliebigen anderen E/A-Baustein ansehen. Der Speicher unterscheidet sich jedoch von E/A-Bausteinen aus folgenden Gründen:

1) **Er arbeitet meistens mit der gleichen Geschwindigkeit wie der Prozessor.**
2) **Er verwendet die gleiche Art von Signalen wie die CPU. Die einzigen Schaltungen, die gewöhnlich zur Anpassung des Speichers an die CPU benötigt werden sind Treiber, Empfänger und Pegelwandler.**
3) **Er benötigt keine speziellen Formate oder irgendwelche Steuersignale ausser einem Schreib/Lese-Impuls.**
4) **Er speichert automatisch die ihm zugesandten Daten.**
5) **Die Wortlänge ist die gleiche wie die des Computers.**

Die meisten E/A-Bausteine besitzen nicht derart bequeme Eigenschaften. Sie können bei Geschwindigkeiten arbeiten, die wesentlich niedriger als die des Prozessors sind. Zum Beispiel kann ein Fernschreiber nur 10 Zeichen pro Sekunde übertragen, während ein langsamer Prozessor 10.000 Zeichen pro Sekunde transferieren kann. **Der Bereich der Geschwindigkeiten ist also sehr breit,** Sensoren können eine Ablesung pro Minute liefern, während Video-Anzeigen oder Floppy-Disks 250.000 Bits pro Sekunde übertragen können. **Ferner benötigen E/A-Bausteine des öfteren kontinuierliche Signale,** (Motoren oder Thermometer), **brauchen eher Strom statt Spannungen** (Fernschreiber), **oder Spannungen, die wesentlich andere Pegel als Signale, die der Prozessor verwendet** (Gas-Entladungs-Anzeigen). E/A-Bausteine können auch spezielle Formate, Übertragungs-Prozeduren oder Steuersignale benötigen. Ihre Wortlänge kann wesentlich kürzer oder wesentlich länger als die Wortlänge des Computers sein. **Alle diese Variationen bedeuten, daß die Entwicklung von E/A-Anordnungen schwierig ist und jeder periphere Baustein seine eigenen speziellen Anpassungs-Probleme besitzt.**

E/A-KATEGORIEN

Wir können jedoch eine allgemeine Beschreibung von Bausteinen und Interfacing-Methoden geben. Wir können die Bausteine grob in drei Kategorien einteilen, basierend auf ihren Datengeschwindigkeiten:

1) **Langsame Bausteine, die ihren Zustand nicht mehr als einmal pro Sekunde ändern.** Die Änderung ihrer Zustände dauert typisch Millisekunden oder länger. Derartige Bausteine umfassen Leuchtanzeigen, Schalter, Relais und zahlreiche mechanische Sensoren und Betätigungsglieder.
2) **Bausteine mit mittlerer Geschwindigkeit, die Daten mit Geschwindigkeiten von 1 bis 10.000 Bits pro Sekunde übertragen.** Derartige Bausteine umfassen Tastaturen, Drucker, Kartenleser, Lochstreifen-Leser und Stanzer, Kassetten, gewöhnliche Übertragungsleistungen und zahlreiche analoge Datenerfassungs-Systeme.
3) **Schnelle Bausteine, die Daten mit Geschwindigkeiten über 10.000 Bits pro Sekunde übertragen.** Derartige Bausteine umfassen Magnetbänder, Magnetplatten, schnelle Zeilendrucker, Kommunikationsleitungen für hohe Geschwindigkeiten und Video-Anzeigen.

ANPASSUNG VON LANGSAMEN BAUSTEINEN

Die Anpassung von langsamen Bausteinen ist einfach. Es sind wenige Steuerbausteine erforderlich, mit Ausnahme für das Multiplexen, das heißt, die Bedienung mehrerer Bausteine von einem Port, wie in den Bilder 11-1 bis 11-4 gezeigt ist. **Die Eingangsdaten** von langsamen Bausteinen **müssen nicht zwischengespeichert werden,** da sie während langer Zeitintervalle stabil bleiben. **Ausgangsdaten müssen** natürlich **zwischengespeichert werden.** Das einzige Problem bei der Eingabe sind die Übergänge, die auftreten, während der Computer die Daten liest. Monostabile Multivibratoren, kreuzgekoppelte Zwischenspeicher, oder Software-Verzögerungsroutinen können die Spannungsprünge glätten.

Ein einzelner Port kann mehrere langsame Bausteine bedienen. Bild 11-1 zeigt einen Demultiplexer, der automatisch die nächsten Ausgangsdaten zum nächsten Baustein dirigiert, indem er die Ausgabe-Operationen zählt. Bild 11-2 zeigt einen Steuerport, der Auswahl-Signal zu einem Demultiplexer liefert. Die Ausgangsdaten können hier in jeder beliebigen Reihenfolge kommen, es ist jedoch ein zusätzlicher Ausgabebefehl erforderlich, um den Zustand des Steuerports zu ändern. Ausgangs-Demultiplexer werden gewöhnlich zur Steuerung mehrerer Anzeigen vom gleichen Ausgangsport verwendet. Die Bilder 11-3 und 11-4 zeigen die gleichen Alternativen für einen Eingangs-Multiplexer. Beachten Sie den Unterschied zwischen Eingabe und Ausgabe bei langsamen Bausteinen:

1) **Eingangsdaten müssen nicht gespeichert werden,** da der Eingangs-Baustein die Daten für eine außerordentlich lange Zeit aufbewahrt, verglichen mit den bei Computer üblichen Zeiten. Ausgangsdaten müssen zwischengespeichert werden, da der Ausgang-Baustein nicht auf Daten reagieren wird, die nur während einiger weniger CPU-Taktzyklen vorliegen.
2) **Eingangs-Spannungssprünge können infolge ihrer Dauer Probleme verursachen. Kurze Ausgangs-Spannungssprünge bewirken keine Probleme, da die Ausgangs-Bausteine (oder die Beobachter) langsam reagieren.**
3) **Die wichtigsten Einschränkungen bei der Eingabe sind die Reaktionszeit und die Empfindlichkeit. Die wesentlichsten Einschränkungen bei der Ausgabe sind die Reaktionszeit und Erkennbarkeit.**

ANPASSUNG VON BAUSTEINEN MIT MITTLERER GESCHWINDIGKEIT

Bausteine mit mittlerer Geschwindigkeit müssen auf irgendeine Weise mit dem Prozessor-Takt synchronisiert werden. Die CPU kann nicht einfach diese Bausteine so behandeln, als ob sie ihre Daten für immer behalten oder Daten zu jeder Zeit empfangen könnten. Stattdessen muß die CPU imstande sein zu bestimmen, wann ein Baustein neue Daten eingegeben hat oder wann er bereit ist, Ausgangsdaten zu empfangen. Sie muß auch eine Möglichkeit besitzen, einem Ausgangs-Baustein mitzuteilen, daß neue Daten verfügbar sind, oder daß die vorhergehenden Eingangsdaten angenommen worden sind. Beachten Sie, daß das periphere Gerät ein anderer Prozessor sein oder einen Prozessor enthalten kann.

QUITTIERUNG

Das standardisierte, nicht getaktete Verfahren ist die Quittierung (Handshake). Hier zeigt der Sender dem Empfänger die Verfügbarkeit von Daten an und überträgt dann die Daten. Der Empfänger vervollständigt die Quittierung, indem er den Empfang der Daten bestätigt. Der Empfänger kann die Situation steuern, indem er anfangs Daten anfordert oder indem er seine Bereitschaft zum Empfang von Daten anzeigt. Der Sender schickt dann die Daten und vervollständigt die Quittierung durch Anzeige, daß die Daten verfügbar sind. In jedem Fall weiß der Sender, daß der Transfer erfolgreich abgeschlossen wurde und der Empfänger weiß, wann die Daten verfügbar sind.

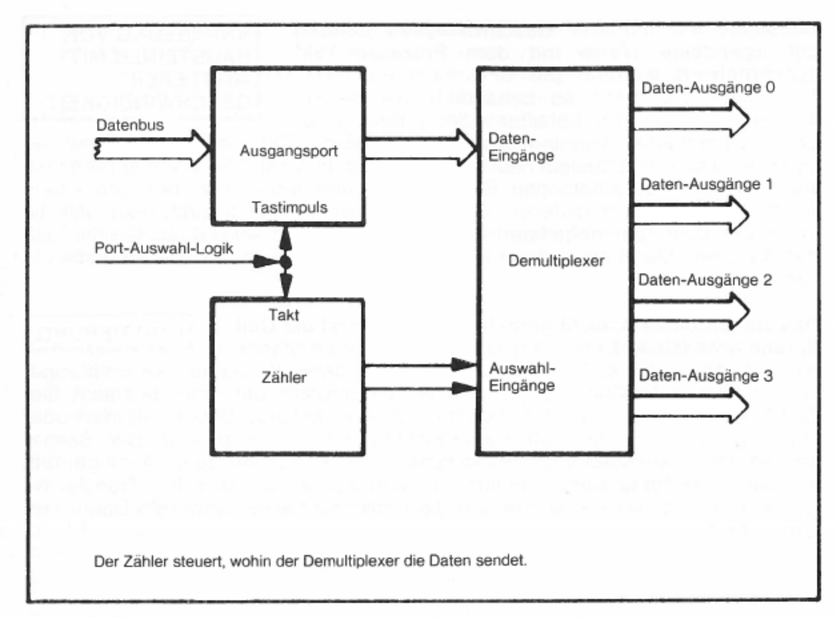


Bild 11-1. Ein Ausgangs-Demultiplexer, gesteuert von einem Zähler.

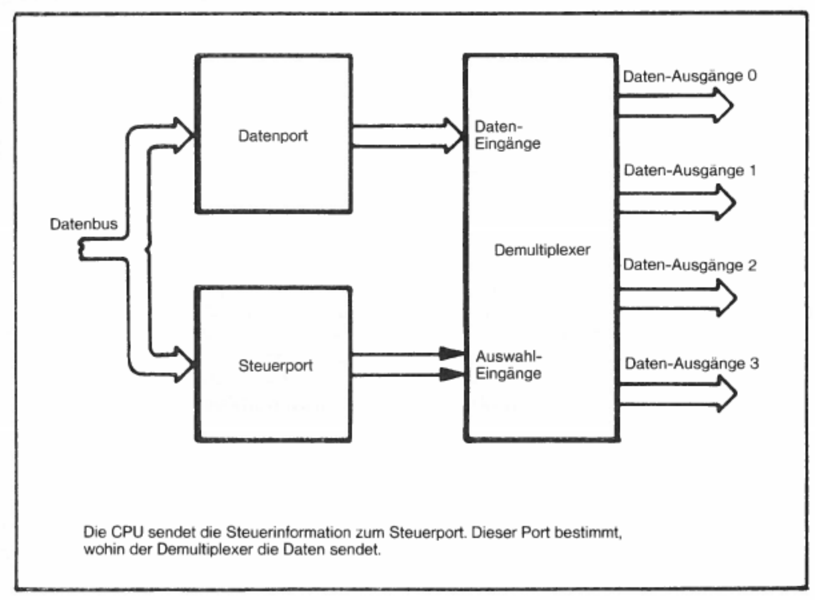


Bild 11-2. Ein Ausgangs-Multiplexer, gesteuert von einem Port.

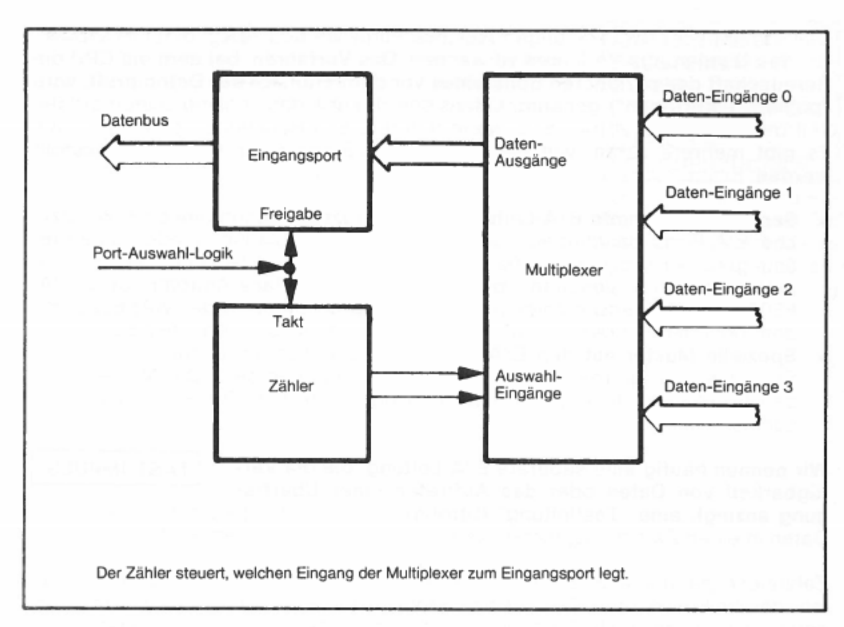


Bild 11-3. Ein Eingangs-Multiplexer, gesteuert von einem Zähler.

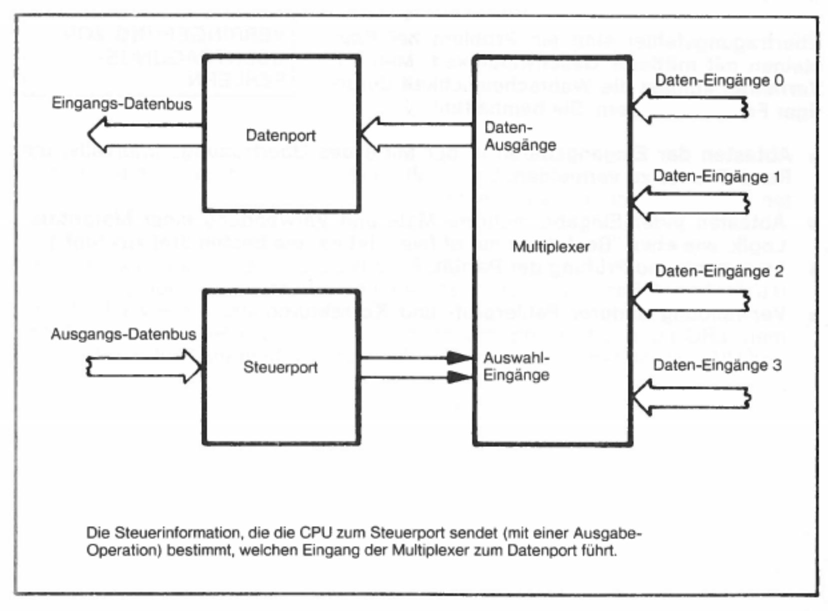


Bild 11-4. Ein Eingangs-Multiplexer, gesteuert von einem Port.

Die Bilder 11-5 und 11-6 zeigen typische Eingabe- und Ausgabe-Operationen, die das Quittierungs-Verfahren verwenden. **Das Verfahren, bei dem die CPU die Bereitschaft des peripheren Bausteines vor dem Transfer der Daten prüft, wird "polling" ("Abfragen") genannt.** Offensichtlich kann das "polling" einen großen Teil der Prozessorzeit belegen, wenn mehrere E/A-Bausteine vorhanden sind. **Es gibt mehrere Arten, wie die Quittierungs-Signale zur Verfügung gestellt werden.** Einige hiervon sind:

- **Seperate bestimmte E/A-Leitungen.** Der Prozessor kann diese als zusätzliche E/A-Ports handhaben oder durch spezielle Leitungen oder Unterbrechungen. Der Prozessor 6502 besitzt keine seriellen E/A-Leitungen. Es sind jedoch derartige Leitungen beim peripheren Interface-Adapter (oder PIA) 6502 verfügbar, sowie beim Versatile Interface-Adapter (oder VIA) 6522 und dem peripheren Interface/Speicher (oder Mehrfunktions)-Baustein 6532.
- **Spezielle Muster auf den E/A-Leitungen.** Dies können einfache Start- und Stop-Bits oder ganze Zeichen oder Zeichengruppen sein. Die Muster müssen leicht vom Hintergrund-Rauschen oder inaktiven Zuständen zu unterscheiden sein.

TAST-IMPULS

Wir nennen häufig eine seperate E/A-Leitung, die die Verfügbarkeit von Daten oder das Auftreten einer Übertragung anzeigt, eine "Tastleitung" (Strobe). Ein Tastleitung kann beispielsweise Daten in einen Zwischenspeicher takten oder Daten von einem Puffer holen.

Zahlreiche periphere Bausteine transferieren Daten in gleichmäßigen Intervallen, das heißt synchron. Das einzige Problem besteht hier im Starten des Vorgangs, indem die erste Eingabe richtig angeordnet wird oder die erste Ausgabe markiert wird. In einigen Fällen liefert der periphere Baustein ein Takt-Signal, von dem der Prozessor Informationen für die zeitliche Steuerung erhalten kann.

VERRINGERUNG VON ÜBERTRAGUNGS-FEHLERN

Übertragungsfehler sind ein Problem bei Bausteinen mit mittlerer Geschwindigkeit. Mehrere Verfahren können die Wahrscheinlichkeit derartiger Fehler verringern. Sie beinhalten:

- **Abtasten der Eingangsdaten in der Mitte des Übertragungs-Intervalls, um Rand-Effekte zu vermeiden.** Das heißt, einen Abstand zu den Enden zu halten, bei denen sich die Daten ändern.
- **Abtasten jeder Eingabe mehrere Male und Verwendung einer Majoritäts-Logik, wie etwa "Best three out of five"[1] (etwa "die besten drei aus fünf")**
- **Erzeugung und Prüfung der Parität.** Ein zusätzliches Bit wird verwendet, das die Zahl der 1-Bits in den korrekten Daten gerade oder ungerade macht.
- **Verwendung anderer Fehlerprüf- und Korrekturcodes,** wie etwa Prüfsummen, LRC (longitudinal reduncy check = longitudinale Redundanz Prüfung) und CRC (cyclic redundancy check = Zyklische Redundanz-Prüfung).[2]

ANPASSUNG VON BAUSTEINEN MIT HOHER GESCHWINDIGKEIT

DIREKTER SPEICHERZUGRIFF

Bausteine mit hoher Geschwindigkeit, die mehr als 10.000 Bits pro Sekunde übertragen, **benötigen spezielle Verfahren.** Die allgemeine Technik besteht im Aufbau eines speziellen Steuergerätes, das Daten direkt zwischen dem Speicher und dem E/A-Baustein transferiert. Dieser Vorgang wird direkter Speicherzugriff (DMA = direct memory access) genannt. Das DMA-Steuergerät muß die CPU von den Bussen fernhalten, Adressen und Steuersignale für den Speicher liefern und die Daten transferieren. Ein derartiges Steuergerät ist ziemlich komplex und besteht gewöhnlich aus 50 bis 100 Chips, obwohl nunmehr LSI-Bausteine erhältlich sind. Die CPU muß anfangs die Adresse und den Datenzähler in das Steuergerät laden, so daß das Steuergerät weiß, wo es zu beginnen hat und wieviel zu übertragen ist.

a) Das periphere Gerät liefert Daten und das Signal "Daten bereit" zum E/A-Abschnitt des Computers.

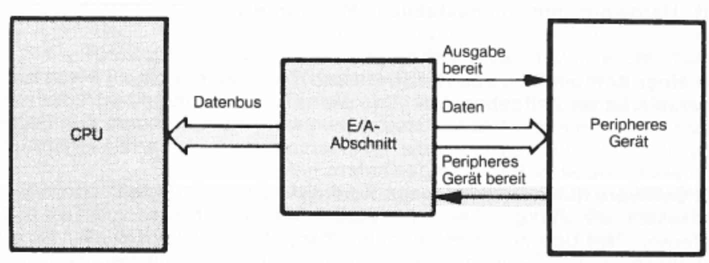
b) Die CPU liest das Signal "Daten bereit" vom E/A-Abschnitt (dies kann eine Hardware-Unterbrechung sein).

c) Die CPU liest die Daten vom E/A-Abschnitt.

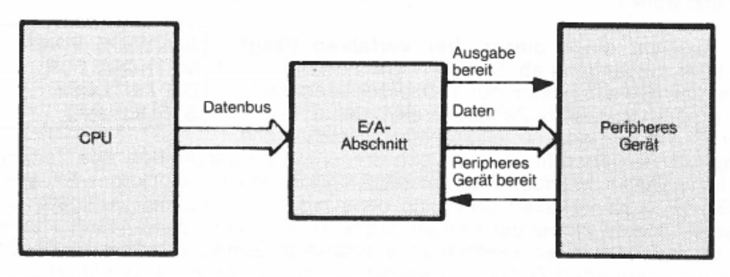
d) Die CPU sendet das Signal "Eingabe-Bestätigung" zum E/A-Abschnitt, der dann ein Signal "Eingabe-Bestätigung" zum peripheren Gerät liefert (dies kann eine Hardware-Verbindung sein).

Bild 11-5. Eine Eingabe-Quittierung.

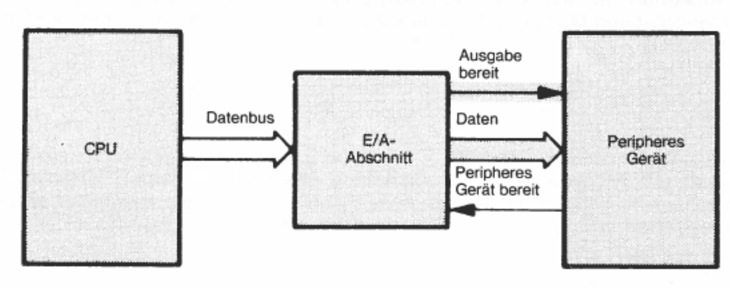
a) Peripheres Gerät liefert ein Signal "Peripheres Gerät bereit" zum E/A-Abschnitt des Computers.



b) Die CPU liest das Signal "Peripheres Gerät bereit" vom E/A-Abschnitt (dies kann eine Hardware-Unterbrechungs-Verbindung sein).



c) Die CPU sendet Daten zum peripheren Gerät.



d) Die CPU sendet das Signal "Ausgabe bereit" zum peripheren Gerät (dies kann eine Hardware-Verbindung sein).

Bild 11-6. Eine Ausgabe-Quittierung.

Zeit-Intervalle (Verzögerungen)

VERWENDUNG VON ZEIT-INTERVALLEN

Eine Aufgabe, der wir während der Besprechung der Eingabe/Ausgabe begegnen werden, **ist die Erzeugung von Zeit-Intervallen mit bestimmten Längen.** Derartige Zeit-Intervalle sind erforderlich, um mechanische Schalter zu entprellen (um ihre unregelmäßigen Spannungssprünge zu glätten), Impulse mit speziellen Längen und Frequenzen für Anzeigen zu liefern, sowie zeitliche Steuerung für Bausteine zu liefern, die Daten gleichmäßig transferieren (beispielsweise ein Fernschreiber, der ein Bit alle 9.1 ms sendet oder empfängt).

METHODEN FÜR DIE ERZEUGUNG VON ZEIT-INTERVALLEN

Wir können Zeit-Intervalle auf verschiedene Arten erzeugen:

1) **Mit Hardware mit monostabilen Multivibratoren.** Diese Bausteine erzeugen einen einzelnen Impuls fester Länge, als Reaktion auf einen Eingangs-Impuls.
2) **Mit einer Kombination von Hardware und Software mit einem flexiblen, programmierbaren Zeitgeber,** wie jene, die im Versatile Interface-Adapter 6522 (der später in diesem Kapitel beschrieben wird) enthalten sind. Die Zeitgeber des 6522 können Zeit-Intervalle unterschiedlicher Länge mit einer Vielzahl von Start- und Stop-Bedingungen liefern.[3]
3) **Mit Software durch Verzögerungs-Routinen.** Diese Routinen verwenden den Prozessor als Zähler. Dies ist möglich, da der Prozessor einen stabilen Referenz-Takt besitzt, wobei jedoch offensichtlich der Prozessor nicht wirtschaftlich verwendet wird. Diese Verzögerungs-Routinen benötigen jedoch keine zusätzliche Hardware und verwenden Prozessor-Zeit, die sonst ungenutzt wäre.

AUSWAHL EINER METHODE FÜR DIE ZEITLOSE STEUERUNG

Die Auswahl eines dieser drei Verfahren hängt von ihrer Anwendung ab. Das Software-Verfahren ist preisgünstig, kann jedoch den Prozessor überlasten. Die programmiebaren Zeitgeber sind relativ teuer, aber leicht anzupassen und imstande, zahlreiche komplexe Aufgaben für die zeitliche Steuerung zu handhaben. Die Zeitgeber, die im Versatile Interface-Adapter 6522 und in den Mehrfunktions-Bausteinen 6503 und 6532 enthalten sind, sind ohne zusätzliche Kosten verfügbar, wenn diese Bausteine verwendet werden. Diese Teile können etwas teurer als einfache Bausteine sein, sind jedoch als vollständige Gehäuse vertretbar. Derartige Teile mit integrierten Zeitgebern werden in zahlreichen Ein-Platin-Mikrocomputern verwendet, einschließlich des KIM, SYM, VIM und AIM-65. Die Verwendung von monostabilen Multiviratoren sollte wenn immer möglich vermieden werden.

Verzögerungs-Routinen

GRUNDLEGENDE SOFTWARE-VERZÖGERUNG

Eine einfache Verzögerungs-Routine arbeitet wie folgt:

Schritt 1) Lade ein Register mit einem spezifizierten Wert.
Schritt 2) Dekrementiere das Register.
Schritt 3) Wenn das Ergebnis von Schritt 2 nicht null ist, wiederhole Schritt 2.

Diese Routine macht nichts weiter, als Zeit benötigen. Der Bertrag der verwendeten Zeit hängt von der Ausführungszeit der verschieden Befehle ab. **Die maximale Länge der Verzögerung ist durch die Größe des Registers begrenzt.** Die gesamte Routine kann jedoch in eine ähnliche Routine plaziert werden, die ein weiteres Register verwendet, usw.

Seien Sie sorgfältig – die tatsächlich verwendete Zeit hängt von der Taktrate ab, mit der der Prozessor läuft, der Geschwindigkeit des Speicherzugriffes und den Betriebs-Bedingungen wie Temperatur, Betriebsspannung und Belastung der Schaltung, die die Geschwindigkeit beeinflussen kann, mit der der Prozessor Befehle ausführt.

TRANSPARENTE VERZÖGERUNGS-ROUTINE

Das folgende Beispiel verwendet die Indexregister X und Y zur Lieferung von Verzögerungen bis zu 255 ms. Die Wahl der Register ist willkürlich. Vielleicht finden Sie auch, daß die Verwendung des Akkumulators oder von Speicherplätzen bequemer ist. Erinnern Sie sich jedoch daran, daß der 6502 keinen ausdrücklichen Befehl zum Dekrementieren des Akkumulators besitzt. Wir könnten eine Routine bilden, die den Inhalt irgendeines Anwender-Registers nicht ändert. Die Sequenz

```
PHP     ;BEWAHRE STATUSREGISTER AUF
PHA     ;BEWAHRE AKKUMULATOR AUF
TXA     ;BEWAHRE INDEXREGISTER X AUF
PHA
TYA     ;BEWAHRE INDEXREGISTER Y AUF
PHA
```

würde den Inhalt aller Register zu Anfang aufbewahren und die Sequenz

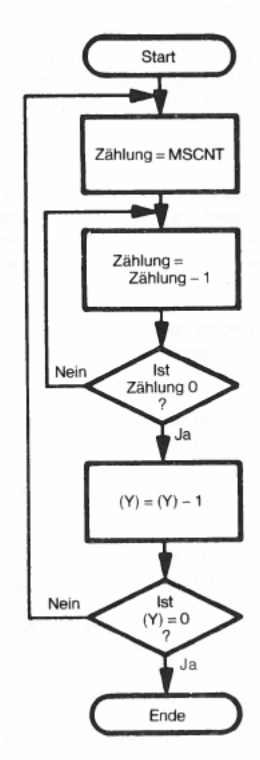
```
PLA     ;SPEICHERE INDEXREGISTER Y ZURÜCK
TAY
PLA     ;SPEICHERE INDEXREGISTER X ZURÜCK
TAX
PLA     ;SPEICHERE AKKUMULATOR ZURÜCK
PLP     ;SPEICHERE STATUSREGISTER ZURÜCK
```

würde die Register am Ende der Routine zurückspeichern. **Ein Unterprogramm, daß keinerlei Register oder Flags beeinflußt, wird "transparent" für das aufrufende Programm genannt.** Die Befehlssequenzen, die die Register aufbewahren und zurückspeichern, müssen natürlich in dem Zeitbudget enthalten sein.

Verzögerungsprogramm:

Zweck: Das Programm liefert eine Verzögerung von 1 ms mal dem Inhalt von Index-Register Y.

Flußdiagramm:



Der Wert von MSCN hängt von der Geschwindigkeit der CPU und dem Speicherzyklus ab.

Quellprogramm:

```
DELAY  LDX   #MSCNT   ;HOLE DIE ZÄHLUNG FÜR 1MS-
                      ;  VERZÖGERUNG
DLY1   DEX            ;ZÄHLUNG = ZÄHLUNG -1
       BNE   DLY1     ;SETZE FORT BIS ZÄHLUNG = 0
       DEY            ;DEKREMENTIERE ANZAHL DER VERBLEI-
                      ;  BENDEN MS
       BNE   DELAY    ;SETZE FORT BIS ANZAHL DER MS = 0
       RTS
```


Objektprogramm: (beginnend im Speicherplatz 0030)

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0030	A2	DELAY	LDX	#MSCNT
0031	MSCNT			
0032	CA	DLY1	DEX	
0033	D0		BNE	DLY1
0034	FD			
0035	88		DEY	
0036	D0		BNE	DELAY
0037	F8			
0038	60		RTS	

Zeit-Budget:

	Befehl	Anzahl der Ausführungen
LDX	#MSCNT	(Y)
DEX		(Y) x MSCNT
BNE	DLY1	(Y) x MSCNT
DEY		(Y)
BNE	DELAY	(Y)
RTS		1

Die gesamte verwendete Zeit sollte gleich (Y) x 1 ms sein. Wenn der Speicher mit voller Geschwindigkeit arbeitet, so benötigen die Befehle die folgenden Anzahl von Taktzyklen:

			Ignorieren der Seitengrenzen
LDX	#MSCNT	2 oder 3	2
DEX	oder DEY	2	2
BNE		2,3 od. 4	2 oder 3
RTS		6	6

Die alternativn Zeiten für LDX #MSCNT hängen davon ab, ob eine Seitengrenze überschritten wird. Die alternativen Zeiten für BNE hängen davon ab, ob die Verzweigung auftritt oder nicht (2), zu einer Adresse auf der gleichen Seite auftritt (3), oder zu einer Adresse auf einer anderen Seite auftritt (4). Eine Seite stellt einen Satz 256 aufeinanderfolgenden Speicherplätzen dar, die die gleichen acht höchstwertigen Bits (oder Seitennummer) in ihren Adressen besitzen. Wir wollen annehmen, daß die Routine so liegt, daß keine Seitengrenzen überschritten werden und wir können die rechte Spalte der letzten Tabelle für die Bestimmung der Zeit verwenden.

Wenn die Befehle Jump-to-Subroutine (JSR) und Return-from-Subroutine (RTS) ignoriert werden (die nur einmal auftreten), benötigt das Programm:

$$(Y) \times (2 + 5 \times MSCNT - 1 + 5) - 1 \text{ Taktzyklen}$$

Die Terme –1 werden durch die Tatsache bewirkt, daß der BNE-Befehl weniger Zeit während der letzten Wiederholung benötigt, wenn der Zähler Null erreicht hat und keine Verzweigung auftritt.

Und daher die Verzögerung ein 1 ms lang zu machen ist

$$5 + 5 \times MSCNT = N_C,$$

wobei N_C die Anzahl der Taktzyklen pro Millisekunde ist. Bei einer Standard-Taktrate von 1 MHz des 6502, $N_C = 1000$ ist daher:

6502 VERZÖGE-GERUNGS-SCHLEI-FEN-KONSTANTE

$5 \times MSCNT = 995$

$MSCNT = 199\ (C7_{16})$ bei einer Taktrate von 1 MHz des 6502.

Eingabe/Ausgabe-Chips des 6502

Die meisten Eingabe/Ausgabe-Abschnitte des 6502 basieren auf LSI-Interface-Chips. Diese Bausteine kombinieren Zwischenspeicher, Puffer, Flipflops und andere Logikschaltungen, die für einen Quittierungsbetrieb und andere einfache Interface-Verfahren erforderlich sind. Sie enthalten zahlreiche Logikverbindungen, wovon bestimmte Sätze entsprechend des Inhaltes von programmierbaren Registern ausgewählt werden können. Der Entwickler hat daher eine Art Äquivalent einer "Schaltungssammlung" zur Verfügung. Die Initialisierungsphase des Programmes plaziert die entsprechenden Werte in Register, um die erforderlichen Logikverbindungen auszuwählen. Ein Eingabe/Ausgabe-Abschnitt, der auf programmierbaren LSI-Interface-Chips basiert, kann zahlreiche unterschiedliche Anwendungen verarbeiten und Änderungen oder Korrekturen können mittels Software ausgeführt werden, anstatt durch eine neuerliche Verdrahtung.

Wir werden die folgenden LSI-Interface-Chips besprechen, die mit dem Mikroprozessor 6502 verwendet werden können:

1) **Der periphere Interface-Adapter 6520.** Dieser Baustein enthält zwei 8-Bit-E/A-Ports und vier individuelle Steuerleitungen und entspricht genau dem 6820, der mit dem Mikrocomputer 6800 verwendet wird.[4]
2) **Der Versatile Interface-Adapter (Vielseitige Interface-Adapter) 6522.** Dieser Baustein enthält zwei 8-Bit-E/A-Ports, vier individuelle Steuerleitungen, zwei 16-Bit-Zähler/Zeitgeber und ein 8-Bit-Schieberegister.
3) **Der periphere Interface/Speicher oder Mehrfunktions (Hilfs)-Baustein 6530.** Dieser Baustein enthält zwei 8-Bit-E/A-Ports, einen 8-Bit-Zähler/-Zeitgeber mit einem Vorteiler, 1024 Bytes ROM und 54 Bytes RAM.
4) **Der periphere Interface/Speicher- oder Mehrfunktions (Hilfs)-Baustein 6532.** Dieser Baustein enthält zwei 8-Bit-E/A-Ports, einen 8-Bit-Zähler/Zeitgeber mit einem Vorteiler und 128 Bytes RAM.

Die folgenden Abkürzungen werden häufig bei der Beschreibung dieser Bausteine verwendet: Der 6520 PIA, 6522 VIA und der 6530 und 6532 RIOT (für ROM oder RAM, I/O, und Timer-Kombination). Unsere E/A-Beispiele, die später in diesem Kapitel behandelt werden, werden alle den VIA 6522 verwenden. Beispiele für die Anwendung des Bausteines 6520 können in dem Buch "6800 Programmieren in Assembler"[5] gefunden werden, wobei diese Beispiele leicht für den 6502 Mikroprozessor adaptiert werden können (erinnern Sie sich an die Vergleiche in den Befehlssätzen in Tabelle 3-6 und 3-7).

Der periphere Interface-Adapter (PIA) 6520

Bild 11-7 zeigt das Blockschaltbild eines PIA. Der Baustein enthält nahezu identisch 8-Bit-Ports: A, der gewöhnlich als Eingangs-Port dient, und B, der gewöhnlich als Ausgangs-Port arbeitet. **Jeder Port enthält:**

PIA-REGISTER UND STEUERLEITUNGEN

- **Ein Daten- oder Peripherie-Register, das entweder der Eingangs- oder Ausgangs-Daten aufbewahrt.** Dieses Register arbeitet im Speicherbetrieb, wenn es für eine Ausgabe verwendet wird, jedoch nicht im Speicherbetrieb, wenn es für eine Eingabe dient.
- **Ein Daten-Richtungsregister.** Die Bits in diesem Register bestimmen, ob die entsprechenden Datenregister-Bits (und Anschlüsse) Eingänge (0) oder Ausgänge (1) sind.
- **Ein Steuer-Register, das die Status-Signale aufbewahrt, die für den Quittierungsbetrieb erforderlich sind, sowie andere Bits, die die Logik-Verbindungen innerhalb des PIA auswählen.**
- **Zwei Steuerleitungen, die durch die Steuer-Register konfiguriert werden.** Diese Leitungen können für die Quittierungs-Signale verwendet werden, wie in den Bildern 11-5 und 11-6 gezeigt ist.

Die Bedeutung der Bits im Datenrichtungs-Register und den Steuer-Registern bezieht sich auf die entsprechende Hardware und ist völlig willkürlich, soweit es den Programmierer für die Assemblersprache betrifft. Man muß sie entweder im Gedächtnis behalten oder kann sie in den entsprechenden Tabellen (Tabellen 11-2 bis 11-6) nachschlagen.

PIA-ADRESSEN

Jeder PIA belegt vier Speicher-Adressen. Die RS (Register Select = Registerauswahl)-Leitungen wählen eines der vier Register aus, wie in Tabelle 11-1 beschrieben ist. Da es in jedem PIA sechs Register gibt (zwei periphere, zwt i Datenrichtungs-Register und zwei Steuer-Register), ist ein weiteres Bit für die Adressierung erforderlich. Bit 2 jedes Steuer-Registers bestimmt, ob sich die andere Adresse auf dieser Seite auf das Datenrichtungs-Register (0) oder das Peripherie-Register (1) bezieht. Diese gemeinsame Verwendung einer externen Adresse bedeutet, daß:

1) Ein Programm das Bit im Steuer-Register ändern muß, um das Register zu verwenden, das momentan nicht adressiert wird.
2) Der Programmierer muß den Inhalt des Steuer-Registers kennen, um zu wissen, welches Register adressiert wird. RESET löscht das Steuer-Register und daher die Adressen im Datenrichtungs-Register.

Tabelle 11-1. Adressierung der internen Register des PIA 5620.

Adressen-Leitungen		Steuer-Register-Bit		Ausgewählte Register
RS1	RSO	CRA-2	CRB-2	
0	0	1	X	Peripheral Register A (Peripheres Register A)
0	0	0	X	Data Direction Register A (Datenrichtungs-Register A)
0	1	X	X	Control Register A (Steuer-Register A)
1	0	X	1	Peripheral Register B (Peripheres Register B)
1	0	X	0	Data Direction Register B (Datenrichtungs-Register B)
1	1	X	X	Control Register B (Steuer-Register B)

X = Entweder 0 oder 1

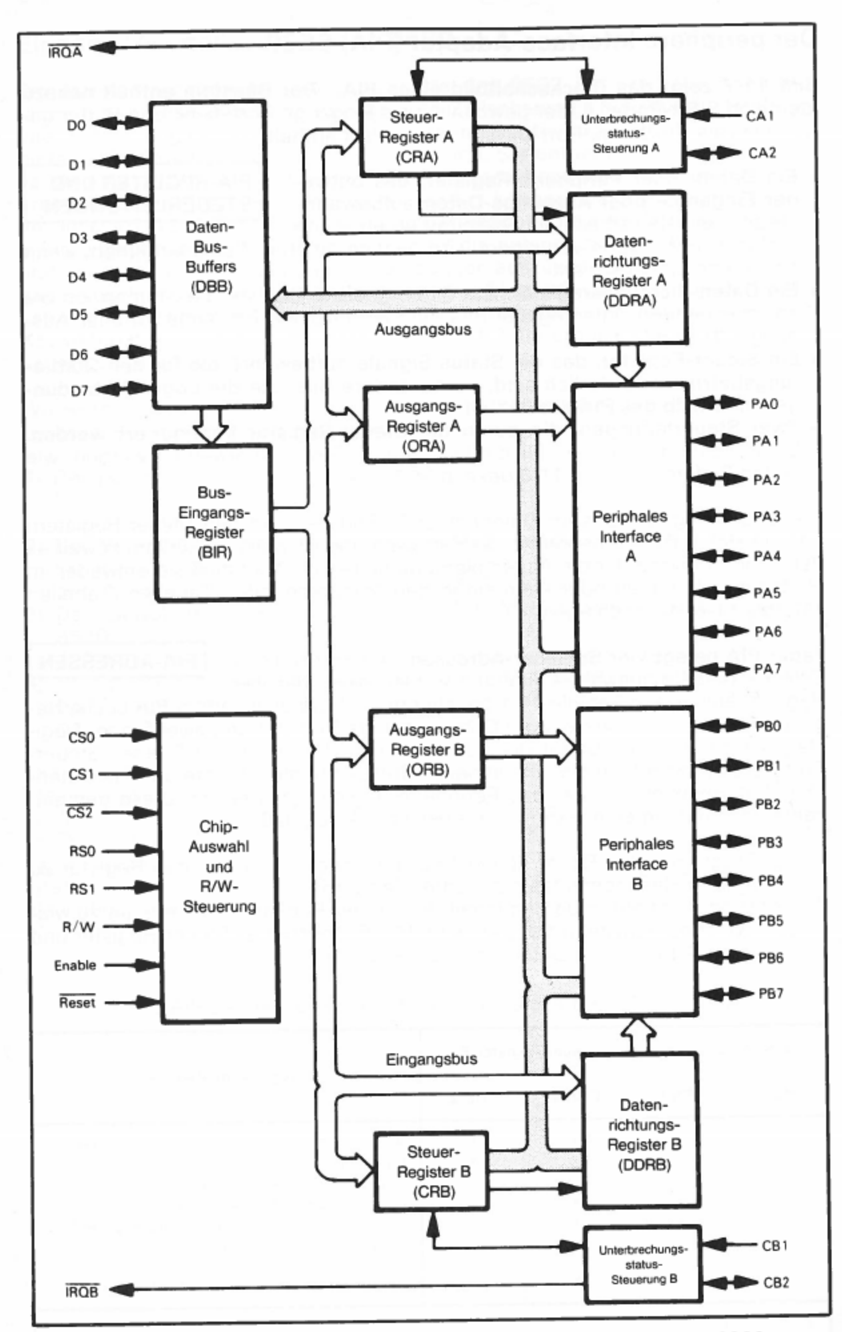


Bild 11-7. Blockschaltbild des Peripheren Interface-Adapters 6820.

PIA-Steuer-Register

STEUER-REGISTER-BITS DES PIA

Tabelle 11-2 zeigt die Organisation der PIA-Steuer-Register. Wir können den allgemeinen Zweck jedes Bits wie folgt beschreiben:

Bit 7: Status-Bit, gesetzt durch Übergänge auf der Steuerleitung 1 und gelöscht durch Lesen des Peripherie-(Daten-)Registers.
Bit 6: Gleich wie Bit 7 mit Ausnahme, daß es durch Übergänge auf der Steuerleitung 2 gesetzt wird.
Bit 5: Bestimmt, ob die Steuerleitung 2 ein Eingang (0) oder Ausgang (1) ist.
Bit 4: Eingang Steuerleitung 2: Bestimmt, ob Bit 6 durch High-auf-Low-Übergänge (0) oder Low-auf-High-Übergänge (1) auf der Steuerleitung 2 gesetzt wird.
Ausgang Steuerleitung 2: Bestimmt, ob Steuerleitung 2 ein Impuls (0) oder ein Pegel(1) ist.
Bit 3: Eingang Steuerleitung 2: Wenn 1, gibt Unterbrechungs-Ausgang von Bit 6 frei.
Ausgang Steuerleitung 2: Bestimmt Endbedingung für Impuls (0 = Quittierungs-Bestätigung, dauert bis zum nächsten Übergang auf der Steuerleitung 1, 1 = kurzer Tast-Impuls, der einen Takt-Zyklus dauert) oder Wert des Pegels.
Bit 2: Wählt Datenrichtungs-Register (0) oder Datenregister (1) aus.
Bit 1: Bestimmt, ob Bit 7 durch High-auf-Low-Übergänge (0) oder Low-auf-High-Übergänge (1) auf der Steuerleitung 1 gesetzt wird.
Bit 0: Wenn 1, gibt Unterbrechungs-Ausgang von Bit 7 des Steuer-Registers frei.

Die Tabelle 11-3 bis 11-6 beschreiben die Bits detaillierter. Da E normalerweise zum ∅2-Takt geführt wird, kann man den "E"-Impuls als "Takt-Impuls" interpretieren.

Tabelle 11-2. Organisation der PIA-Steuer-Register

	7	6	5	4	3	2	1	0
CRA	IRQA1	IRQA2	CA2-Steuerung			DDRA-Zugriff	CA1-Steuerung	
	7	6	5	4	3	2	1	0
CRB	IRQB1	IRQB2	CB2-Steuerung			DDRB-Zugriff	CB1-Steuerung	

Tab. 11-3. Steuerung der Unterbrechungs-Eingänge CA1 u. CA2 des PIA 6520

CRA-1 (CRB-1)	CRA-0 (CRB-0)	Unterbrechungs-Eingang CA1 (CB1)	Unterbrechungs-Flag CRA-7 (CRB-7)	MPU-Unterbrechungs-Anforderung $\overline{IRQA}$ ($\overline{IRQB}$)
0	0	↓ Aktiv	High gesetzt bei ↓ von CA1 (CB1)	Freigegeben – $\overline{IRQ}$ bleibt High
0	1	↓ Aktiv	High gesetzt bei ↓ von CA1 (CB1)	Geht auf Low, wenn das Unterbrechungs-Flag-Bit CRA-7 (CRB-7) auf High geht
1	0	↑ Aktiv	High gesetzt bei ↑ von CA1 (CB1)	Freigegeben – $\overline{IRQ}$ bleibt High
1	1	↑ Aktiv	High gesetzt bei ↑ von CA1 (CB1)	Geht auf Low, wenn das Unterbrechungs-Flag-Bit CRA-7 (CRB-7) auf High geht

Anmerkungen:

1. ↑ zeigt einen positiven Übergang an (Low auf High)
2. ↓ zeigt einen negativen Übergang an (High auf Low)
3. Das Unterbrechungs-Flag-Bit CRA-7 wird durch ein MPU-Lesen des Datenregisters A gelöscht, und CRB-7 wird durch ein MPU-Lesen des Datenregisters B gelöscht.
4. Wenn CRA-0 (CRB-0) beim Auftreten einer Unterbrechung Low ist (Unterbrechung gesperrt) und später auf High gebracht wird, tritt $\overline{IRQA}$ ($\overline{IRQB}$) auf, nachdem CRA-0 (CRB-0) auf "Eins" geschrieben wurde.

Tab. 11-4. Steuerung der Unterbrechungs-Eingänge CA2 u. CB2 des PIA 6520.

CRA-5 (CRB-5)	CRA4 (CRB-4)	CRA-3 (CRB-3)	Unterbrechungs-Eingang CRA-7 (CRB-7)	Unterbrechungs-Flag CA1 (CB1)	MPU-Unterbrechungs-Anforderung $\overline{IRQA}$ ($\overline{IRQB}$)
0	0	0	↓ Aktiv	High gesetzt bei ↓ von CA2 (CB2)	Freigegeben – $\overline{IRQ}$ bleibt High
0	0	1	↓ Aktiv	High gesetzt bei ↓ von CA2 (CB2)	Geht auf Low, wenn das Unterbrechungs-Flag-Bit CRA-6 (CRB-6) auf High geht
0	1	0	↑ Aktiv	High gesetzt bei ↑ von CA2 (CB2)	Freigegeben – $\overline{IRQ}$ bleibt High
0	1	1	↑ Aktiv	High gesetzt bei ↑ von CA2 (CB2)	Geht auf Low, wenn das Unterbrechungs-Flag-Bit CRA-6 (CRB-6) auf High geht

Anmerkungen:

1. ↑ zeigt einen positiven Übergang an (Low auf High)
2. ↓ zeigt einen negativen Übergang an (High auf Low)
3. Das Unterbrechungs-Flag-Bit CRA-6 wird durch ein MPU-Lesen des Datenregisters A gelöscht, und CBR-6 wird durch ein MPU-Lesen des Datenregisters B gelöscht.
4. Wenn CRA-3 (CRB-3) beim Auftreten einer Unterbrechung Low ist (Unterbrechung gesperrt) und später auf High gebracht wird, tritt $\overline{IRQA}$ ($\overline{IRQB}$) auf, nachdem CRA-3 (CRB-3) auf "Eins" geschrieben wurde.

Tabelle 11-5. Steuerung der CB2-Ausgangs-Leitung des PIA 6520.

CRB-5	CRB-4	CRB-3	CB2	
			Gelöscht	Gesetzt
1	0	0	Low beim positiven Übergang des ersten E-Impulses, der einer MPU-Schreib-Operation des "B"-Daten-Registers folgt.	High, wenn das Unterbrechungs-Flag-Bit CRB-7 durch einen aktiven Übergang des CB1-Signals gesetzt wird.
1	0	1	Low beim positiven Übergang des ersten E-Impulses nach einer MPU-Schreib-Operation des "B"-Daten-Registers.	High an der positiven Flanke des ersten "E"-Impulses, der einem "E"-Impuls folgt, der auftrat, während das Teil nicht gewählt war.
1	1	0	Low, wenn CRB-3 auf Low geht, als Ergebnis eines MPU-Schreibens in das Steuer-Register "B".	Immer Low, so lange CRB-3 Low ist. Wird bei einem MPU-Schreiben in das Steuer-Register "B" auf High gehen, das CRB-3 auf "Eins" ändert.
1	1	1	Immer High, so lange CRB-3 High ist. Wird gelöscht, wenn ein MPU-Schreiben des Steuer-Registers "B" in einem Löschen von CRB-3 auf "null" resultiert.	High, wenn CRB-3 auf High geht, als Ergebnis eines MPU-Schreibens in das Steuer-Register "B".

Tabelle 11-6. Steuerung der CA2-Ausgangs-Leitung des PIA 6520.

CRA-5	CRA-4	CRA-3	CA2	
			Gelöscht	Gesetzt
1	0	0	Low bei einem negativen Übergang von E nach einer MPU-Lese-Operation der "A"-Daten.	High, wenn das Unterbrechungs-Flag-Bit CRA-7 durch einen aktiven Übergang des CA1-Signals gesetzt wird.
1	0	1	Low bei einem negativen Übergang von E nach einer MPU-Lese-Operation der "A"Daten	High bei der negativen Flanke des ersten "E"-Impulses, der während der Zeit auftritt, in der keine Auswahl vorliegt.
1	1	0	Low, wenn CRA-3 auf Low geht, als Ergebnis eines MPU-Schreibens zum Steuer-Register "A".	Immer Low, so lange CRA-3 Low ist. Wird auf High gehen bei einem MPU-Schreiben zum Steuer-Register "A", das CRA-3 auf "Eins" ändert.
1	1	1	Immer High, so lange CRA-3 High ist. Wird bei einem MPU-Schreiben zum Steuer-Register "A" gelöscht, das CRA-3 auf "Null" löscht.	High, wenn CRA-3 auf High als Ergebnis eines MPU-Schreibens in das Steuer-Register "A" geht.

Konfigurieren des PIA

SCHRITTE BEI DER KONFIGURIERUNG EINES PIA

Das Programm muß die Logik-Verbindung im PIA auswählen, bevor dieser verwendet wird. Diese Auswahl (oder Konfiguration) ist gewöhnlich ein Teil der Start-Routine. Die Schritte bei der Konfiguration sind:

1) Adressiere das Datenrichtungs-Register durch Löschen von Bit 2 des Steuer-Registers. Da das Reset-Signal alle internen Register löscht, ist dieser Schritt in der Gesamt-Startroutine nicht notwendig.
2) Stelle die Richtungen der E/A-Anschlüsse durch Laden des Datenrichtungs-Registers ein.
3) Wähle die erforderlichen Logik-Verbindungen im PIA durch Laden des Steuer-Registers aus. Setze Bit 2 des Steuer-Registers, um das Datenregister zu adressieren.

Schritt 1 kann wie folgt ausgeführt werden:

```
        LDA   #0          ;LÖSCHE PIA-STEUERREGISTER
        STA   PIACR
oder
        LDA   PIACR
        AND   #%11111011  ;WÄHLE DATENRICHTUNGS-REGISTER
```

Sobald das Programm Schritt 1 ausgeführt hat, ist Schritt 2 einfach eine Angelegenheit des Löschens jedes Eingangs-Bits und das Setzen jedes Ausgangs-Bits im Datenrichtungs-Register. Einige einfache Beispiele sind:

```
1)      LDA   #0          ;ALLE LEITUNGEN EINGÄNGE
        STA   PIADDR

2)      LDA   #$FF        ;ALLE LEITUNGEN AUSGÄNGE
        STA   PIADDR

3)      LDA   #$F0        ;SETZE LEITUNGEN 4-7 ALS AUSGÄNGE,
                          ;  0-3 ALS EINGÄNGE
        STA   PIADDR
```


Schritt 3 ist offensichtlich der schwierige Teil der Konfiguration, da es die Auswahl der Logik-Verbindungen in der PIA beinhaltet. Man muß sich hierbei an einige Punkte erinnern:

1) Bit 6 und 7 des Steuer-Registers werden durch Übergänge auf den Steuerleitungen gesetzt und durch Lesen des Datenregisters gelöscht. Man kann diese Bits nicht durch Schreiben von Daten in das Steuer-Register ändern.
2) bit 2 des Stuer-Registers muß gesetzt werden, um das Datenregister zu adressieren.
3) Bit 1 bestimmt, welche Impuls-Flanke Bit 7 setzen wird. Bit 1 ist 0 für einen High-auf-Low-Übergang. Bit 1 ist 1 für einen Low-auf-High-Übergang.
4) Bit 0 ist die Unterbrechungs-Freigabe für Steuerleitung 1. Erinnern wir uns daran, daß es zur Freigabe von Unterbrechungen gesetzt werden muß, im Gegensatz zum Unterbrechungs-Bit des 6502, das zur Freigabe von Unterbrechungen gelöscht werden muß. Kapitel 12 beschreibt Unterbrechungen detaillierter.
5) Bit 5 muß gesetzt werden, wenn Steuerleitung 2 ein Ausgang sein soll. Die Bits 3 und 4 bestimmen dann, wie die Steuerleitung 2 arbeitet. Erinnern wir uns daran, daß die Seiten A und B differieren, da die Seite A nur einen Lese-Impuls erzeugen kann, während die Seite B nur einen Schreib-Impuls bilden kann. Sobald die Impuls-Option gewählt wurde, folgen die Impulse automatisch jedem Lesen des Datenregisters A oder Schreiben des Datenregisters B. Man muß jede Seite jedes PIA im Startprogramm konfigurieren.

BEISPIELE FÜR DIE PIA-KONFIGURATION

PIA-KONFIGURATIONS-BEISPIELE

1) Ein einfacher Eingangs-Port ohne Steuerleitungen:

```
LDA   #0          ;LÖSCHE STEUER-REGISTER
STA   PIACR
STA   PIADDR      ;MACHE ALLE LEITUNGEN ZU EINGÄNGEN
LDA   #%00000100  ;WÄHLE DATENREGISTER
STA   PIACR
```

Bit 2 des Steuer-Registers muß gesetz werden, um das Datenregister zu adressieren. Die gleiche Sequenz kann verwendet werden, wenn eine High-auf-Low-Flanke auf der Steuerleitung 1 ein "Daten-Bereit" (Data Ready) oder "Peripherie-Bereit" (Peripheral Ready) anzeigt.

2) Ein einfacher Ausgangs-Port ohne Steuerleitungen (wie er für einen Satz einzeln LED-Anzeigen benötigt wird):

```
LDA   #0          ;LÖSCHE STEUER-REGISTER
STA   PIACR
LDA   #$FF        ;MACHE ALLE LEITUNGEN ZU AUSGÄNGEN
STA   PIADDR
LFDA  #%00000100  ;WÄHLE DATENREGISTER
STAA  PIACR
```

3) Ein Eingangs-Port, der DATA READY mit einem Low-auf-High-Übergang (positiver Übergang anzeigt):

```
LDA   #0          ;LÖSCHE STEUER-REGISTER
STA   PIACR
STA   PIADDR      ;MACHE ALLE LEITUNGEN ZU EINGÄNGEN
LDA   #%00000110  ;MACHE "DATA-READY" AKTIV LOW-ZU-
                  ;  HIGH
STA   PIACR
```

Die Leitungen DATA-READY oder DATA-AVAILABLE werden zu den Steuerleitungen CA1 oder CB1 geführt. Bit 1 des Sturregisters wird so gesetzt, daß Low-auf-High-Übergänge auf der Steuerleitung 1 erkannt werden. Diese Konfiguration ist geeignet für die meisten codierten Tastaturen.

4) Ein Ausgangs-Port, der einen kurzen Impuls zur Anzeige von DATA READY und OUTPUT READY erzeugt, (dies könnte zum Multiplexen von Anzeigen oder zur Lieferung eines Signales DATA AVAILABLE zu einem Drucker verwendet werden):

```
LDA   #0          ;LÖSCHE STEUER-REGISTER
CLR   PIACR
LDA   #$FF        ;MACHE ALLE LEITUNGEN ZU AUSGÄNGEN
STA   PIADDR
LDA   %00101100   ;STEUERLEITUNG 2 = KURZER TASTIMPULS
STA   PIACR
```


Bit 5 = 1, um die Steuerleitung 2 zu einem Ausgang zu machen, Bit 4 = 0, um einen Impuls zu erzeugen, und Bit 3 = 1, um einen kurzen aktiven Low-Impuls (eine Taktperiode lang) zu bilden. Der Tast-Impuls wird automatisch jedem Befehl folgen, der Daten in die B-Seite des PIA schreibt. Zum Beispiel wir der Befehl

```
STA   PIADRB
```

sowohl Daten transferieren, als auch einen Tast-Impuls bewirken. Die A-Seite jedoch wird einen Tast-Impuls nur nach einer Lese-Operation erzeugen. Die Sequenz

```
STA   PIADRA      ;SCHREIBE DATEN
LDA   PIDR        ;ERZEUGE EINEN AUSGANGS-TAST-IMPULS
```

wird sowohl Daten tranferieren als auch einen Tast-Impuls bewirken. Der LDA-Befehl ist ein "Leer-Lesen" (dummy read). Er hat keinen anderen Einfluß, als den Tast-Impuls zu bewirken (und einige Zeit zu verwenden). Andere Befehle neben LDA könnten ebenfalls verwendet werden (zählen Sie einige auf).

5) Ein Eingangs-Port mit einem Bestätigungs-Impuls für einen Quittierungs-Eingang der dazu verwendet werden kann, um einem peripheren Gerät mitzuteilen, daß die vorausgehenden Daten angenommen wurden (und der Computer bereit für weitere ist):

```
LDA   #0          ;LÖSCHE STEUER-REGISTER
STA   PIACR
STA   PIADDR      ;MACHE ALLE LEITUNGEN ZU EINGÄNGEN
LDA   #%00100100  ;STEUERLEITUNG 2 = QUITTIERUNGS-
                  ;  BESTÄTIGUNG
STA   PIACR
```

Bit 5 = 1, um die Steuerleitung 2 zu einem Ausgang zu machen, Bit 4 = 0 für einen Impuls, und Bit 3 = 0, um eine Quittierung mit aktiv Low zu bilden, die Low bis zum nächsten aktiven Übergang auf der Steuerleitung 1 bleibt. Der Bestätigung wird automatisch eine Lese-Operation auf der A-Seite des PIA folgen. Zum Beispiel wird der Befehl

```
LDA   PIADRA
```

sowohl die Daten lesen, wie auch die Bestätigung bewirken. Die B-Seite wird jedoch eine Bestätigung nur nach einer Schreib-Operation erzeugen. Die Sequenz

```
LDA   PIADRB      ;LIES DATEN
STA   PIADRB      ;ERZEUGE BESTÄTIGUNG
```

wird sowohl die Daten lesen, als auch eine Bestätigung erzeugen. Der STA-Befehl ist ein "dummy-write". Er hat keinen anderen Effekt, als die Bestätigung zu erzeugen (und einige Zeit zu verwenden). Beachten Sie, daß die Reihenfolge der Sequenz gegenüber dem vorhergehenden Beispiel umgekehrt wurde. Diese Konfiguration ist geeignet für zahlreiche Bildschirm-Terminals, die eine vollständige Quittierung benötigen.

6) Ein Ausgangsport mit einem zwischengespeicherten Null-Steuer-Bit (zwischengespeicherter individueller Ausgang oder Pegel-Ausgang). Ein derartiger Ausgang kann zum Ein- oder Ausschalten des peripheren Gerätes verwendet werden, oder zur Stuerung seiner Betriebsart.

```
LDA   #0           ;LÖSCHE STEUER-REGISTER
STA   PIACR
LDA   #$FF         ;MACHE ALLE LEITUNGEN ZU AUSGÄNGEN
STA   PIADDR
LDA   #%00110100   ;STEUERLEITUNG 2 = GESPEICHERTER
                   ;  NULL-PEGEL
STA   PIACR
```

Bit 5 = 1, um die Steuerleitung 2 zu einem Ausgang zu machen, Bit 4 = 1, um einen Pegel oder ein gespeichertes Bit zu bilden, und Bit 3 = 0, um den Pegel null zu machen. Dieses Bit wird nicht durch die Operationen am Datenregister beeinflußt. Sein Wert kann durch Ändern des Wertes von Bit 3 verändert werden, das heißt:

```
LDA   PIACR
ORA   #%00001000   ;MACHE PEGEL EINS
STA   PIACR

LDA   PIACR
AND   #%11110111   ;MACHE PEGEL NULL
STA   PIACR
```

Man kann diese Konfiguration verwenden, um Tast-Impulse mit aktiv High zu erzeugen oder Impulse zu liefern, bei denen die Länge durch die Software gesteuert wird.

VERWENDUNG DES PIA ZUM TRANSFERIEREN VON DATEN

PIA-EINGABE/ AUSGABE

Sobald der PIA konfiguriert wurde, kann man seine Datenregister wie jeden anderen Speicherplatz verwenden. Die einfachsten Befehle für den Datentransfer sind:

"Lade Akkumulator" transferiert 8 Datenbits von dem spezifizierten Eingangs-Anschlüssen zum Akkumulator.

"Speichere Akkumulator" transferiert 8 Datenbits von einem Akkumulator zu den spezifizieten Ausgangs-Anschlüssen.

Man muß sehr sorgfältig in Situatione vorgehen, bei denen Eingangs- und Ausgangs-Ports sich nicht wie Speicherplätze verhalten. Beispielsweise hat es häufig keinen Sinn, Daten in Eingangs-Ports zu schreiben oder Daten von Ausgangs-Ports zu lesen. Besonders sorgfältig muß man vorgehen, wenn der Eingangs-Port nicht zwischengespeichert ist oder wenn der Ausgangs-Port nicht gepuffert ist.

Andere Befehle, die Daten zu oder vom Speicher transferieren, können auch als E/A-Befehle dienen. Typische Beispiele sind:

Bit-Test, der das Nullflag setzt, als ob die Werte eines Satzes von Eingangs-Pins logisch mit dem Inhalt des Akkumulators UNDiert worden wären. Das Vorzeichen- (Negativ)Flag wird auf den Wert von Bit 7 des Eingangsports und das Überlauf-Flag wird auf den Wert von Bit 6 des Eingangsports gesetzt. **Dieser Befehl liefert einen einfachen Weg, um die PIA-Statusflags zu testen.** Das heißt, der Befehl

```
BIT   PIACR
```

setzt das Vorzeichen-Flag auf den Wert des Steuerregister-Bits 7 (der Status-Zwischenspeicher für Steuerleitung 1) und das Überlauf-Flag auf den Wert des Steuerregister-Bit 6 (der Status-Zwischenspeicher für Steuerleitung 2).

"Compare" setzt die Flags, als ob sie Werte eines Satzes von Eingangs-Anschlüssen wären, die vom Inhalt des Akkumulators subtrahiert wurden.

Hier muß man sich auch die physikalischen Grenzen der E/A-Ports vor Augen halten. Besonders sorgfältig muß man bei Befehlen für Verschieben, Komplementieren, Inkrementieren und Dekrementieren sein, die sowohl Lese- wie Schreibzyklen beinhalten.

Man kann die Bedeutung einer sorgfältigen Dokumentation nicht nachdrücklich genug betonen. Häufig können komplexe E/A-Transfers in Befehlen versteckt sein, die keine offensichtlichen Funktionen besitzen. Man muß den Zweck derartiger Befehle sehr sorgfältig beschreiben. Beispielsweise könnte jemand versucht sein, die "dummy read"- und "write"-Operationen, die früher erwähnt wurden, zu entfernen, da sie scheinbar keine Funktion besitzen.

Bit 7 des PIA-Steuer-Registers dient häufig als Status-Bit, wie etwa "Data Ready" oder "Peripheral Ready". Man kann ihre Werte mit einer der folgenden Sequenzen überprüfen:

PIA-STATUS-BITS

```
        LDA    PIACR      ;IST READY-FLAG 1?
        BMI    DEVRDY     ;JA, BAUSTEIN BEREIT

        BIT    PIACR      ;IST READY-FLAG 1?
        BMI    DEVRDY     ;JA, BAUSTEIN BEREIT
```

Beachten Sie, daß man Schiebe-Befehle nicht verwenden sollte, da Sie den Inhalt des Steuer-Registers verändern werden (warum?). Das folgende Programm wird darauf warten, daß das Ready-Flag auf High geht:

```
WAITR   BIT    PIACR      ;IST READY-FLAG 1?
        BPL    WAITR      ;NEIN, WARTE
```

Wie würden Sie diese Programme ändern, so daß Sie Bit 6 anstatt Bit 7 prüfen?

Der einzige Weg zum Löschen von Bit 7 (oder Bit 6) ist das Lesen des Datenregisters. Ein "dummy read" wird erforderlich sein, wenn eine Lese-Operation normalerweise nicht Teil der Reaktion auf das gesetzte Bit ist. Wenn der Port für Ausgabe verwendet wird, so wird die Sequenz

```
        STA    PIADR      ;SENDE DATEN
        LDA    PIADR      ;LÖSCHE LESE-FLAG
```

diese Aufgabe erfüllen. Beachten Sie, daß hier das "dummy read" auf jeder Seite des PIA erforderlich ist. Der Testbefehl kann auch den Tast-Impuls ohne irgendwelche anderen Änderungen löschen, mit Ausnahme der Flags. Seien Sie besonders sorgfältig in Fällen, bei denen die CPU nicht bereit für die Eingabe von Daten ist, oder keine Ausgangsdaten zu senden hat.

DER VERSATILE INTERFACE-ADAPTER (VIA) 6522

Der "Vielseitige Interface-Adapter" 6522 ist eine verbesserte Version des peripheren Interface-Adapters 6520.[6,7,8]

Der VIA 6522 enthält folgendes (siehe Blockschaltbild in Bild 11-8):

6522 VIA-FUNKTIONEN

1) **Zwei 8-Bit-E/A-Ports (A und B).** Jeder Anschluß kann individuell ausgewählt werden, um entweder als Eingang oder als Ausgang zu dienen.

2) **Vier Status- und Steuerleitungen** (zwei jedem Port zugeordnet).

3) **Zwei 16-Bit-Zähler/Zeitgeber,** die zur Erzeugung oder zum Zählen von Impulsen verwendet werden können. Diese Zeitgeber können einzelne Impulse oder eine kontinuierliche Serie von Impulsen erzeugen.

4) **Ein 8-Bit-Schieberegister,** das Daten seriell/parallel umwandeln kann.

5) **Unterbrechungslogik** (wird in Kapitel 12 beschrieben), so daß E/A auf einer unterbrechungsgesteuerten Basis ausgeführt werden kann.

Der vielseitige Interface-Adapter liefert die Funktionen des PIA, plus zweier 16-Bit-Zähler/Zeitgeber und einem 8-Bit-Schieberegister. Wir wollen die Verwendung der Zähler/Zeitgeber später in diesem Kapitel behandeln. Das Schieberegister liefert eine einfache serielle E/A-Möglichkeit, die nur gelegentlich von Nutzen ist. Wir wollen sie daher nicht weiter besprechen.

VIA-ADRESSEN

Jeder VIA belegt 16 Speicher-Adressen. Die RS-Leitungen (Register Select) wählen die verschiedenen internen Register aus, wie in Tabelle 11-7 beschrieben ist. **Die Art und Weise, in der ein VIA arbeitet, wird durch den Inhalt von vier Registern bestimmt.**

VIA-REGISTER UND STEUERLEITUNGEN

1) **Datenrichtungs-Register A (DDRA = Data Direction Register A)** bestimmt, ob die Anschlüsse an Port A Eingänge oder Ausgänge sind.

2) **Datenrichtungs-Register B (DDRB)** bestimmt, ob die Anschlüsse an Port B Eingänge oder Ausgänge sind.

3) **Das periphere Steuerregister (PCR = Peripheral Control Register)** bestimmt, welche Polarität des Übergangs (Anstiegsflanke oder abfallende Flanke) auf den Eingangs-Statusleitungen (CA1 und CB1) erkannt wird, und wie die anderen Statusleitungen (CA2 und CB2) arbeiten. Bild 11-9 beschreibt die Bä-Zuweisungen im peripheren Steuerregister. Wie gewöhnlich sind die Funktionen und Bit-Positionen willkürlich vom Hersteller ausgewählt worden. Beachten Sie, daß das periphere Steuerregister des 6522 nicht Status-Bits (zwischengespeichert) enthält, wie die Steuerregister des 6520. Diese Bits liegen im seperaten Unterbrechungsflag-Register (siehe Bild 11-11).

4) **Das Hilfssteuer-Register (ACR = Auxiliary Control Register)** bestimmt, ob die Datenports zwischengespeichert werden und wie die Zeitgeber und das Schieberegister erarbeiten. Diese Funktionen sind im PIA 6520 nicht enthalten. Bild 11-10 beschreibt die Bit-Zuweisungen im Hilfs-Steuerregister.

Beachten sie, daß es ein Datenrichtungs-Register für jede Seite gibt, jedoch nur ein Steuerregister (anders als beim 6520, der ein seperates Steuerregister für jede Seite besitzt). Die Ports A und B sind im wesentlichen identisch. Ein wichtiger Unterschied besteht darin, daß Port B Darlington-Transistoren ansteuern kann, die zum Treiben von Spulen und Relais verwendet werden. Wir wollen Port A für Eingabe und Port B für Ausgabe in unseren Beispielen später in diesem Kapitel verwenden.

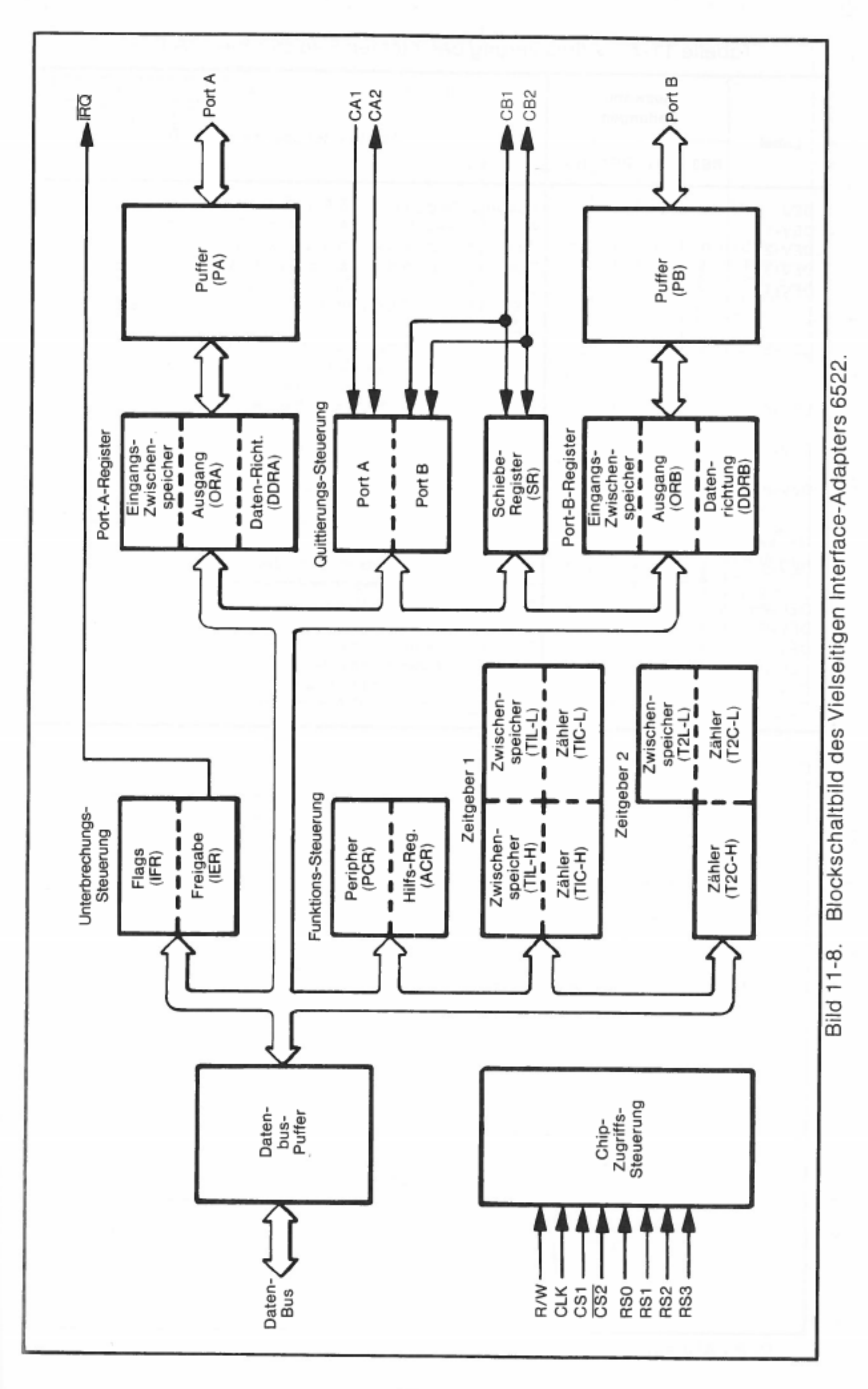


Bild 11-8. Blockschaltbild des Vielseitigen Interface-Adapters 6522.

Tabelle 11-7. Adressierung der internen Register des VIA 6522.

Label	Auswahl-Leitungen				Adressierter Speicherplatz
	RS3	RS2	RS1	RS0	
DEV	0	0	0	0	Ausgangs-Register für E/A-Port B
DEV+1	0	0	0	1	Ausgangs-Register für E/A-Port A, mit Quittierung
DEV+2	0	0	1	0	Datenrichtungs-Register des E/A-Ports B
DEV+3	0	0	1	1	Datenrichtungs-Register des E/A-Ports A
DEV+4	0	1	0	0	Lies niederwertiges Byte des Zählers von Zeitgeber 1 Schreibe zum niederwertigen Byte des Zwischenspeichers von Zeitgeber 1
DEV+5	0	1	0	1	Lies hochwertiges Byte des Zählers von Zeitgeber 1 Schreibe zum hochwertigen Byte des Zwischenspeichers von Zeitgeber 1 und initiiere Zählung
DEV+6	0	1	1	0	Greife auf niederwertiges Byte des Zwischenspeichers von Zeitgeber 1 zu
DEV+7	0	1	1	1	Greife auf hochwertiges Byte des Zwischenspeichers von Zeitgeber 1 zu
DEV+8	1	0	0	0	Lies niederwertiges Byte von Zeitgeber 2 und lösche Zähler-Unterbrechung Schreibe zu niederwertigem Byte von Zeitgeber 2, lösche jedoch Unterbrechung nicht
DEV+9	1	0	0	1	Greife auf hochwertiges Byte von Zeitgeber 2 zu, lösche Zähler-Unterbrechung beim Schreiben
DEV+A	1	0	1	0	Serielles E/A-Schiebe-Register
DEV+B	1	0	1	1	Hilfs-Steuer-Register
DEV+C	1	1	0	0	Peripheres Steuer-Register
DEV+D	1	1	0	1	Unterbrechungs-Flag-Register
DEV+E	1	1	1	0	Unterbrechungs-Freigabe-Register
DEV+F	1	1	1	1	Ausgangs-Register für E/A-Port A, ohne Quittierung

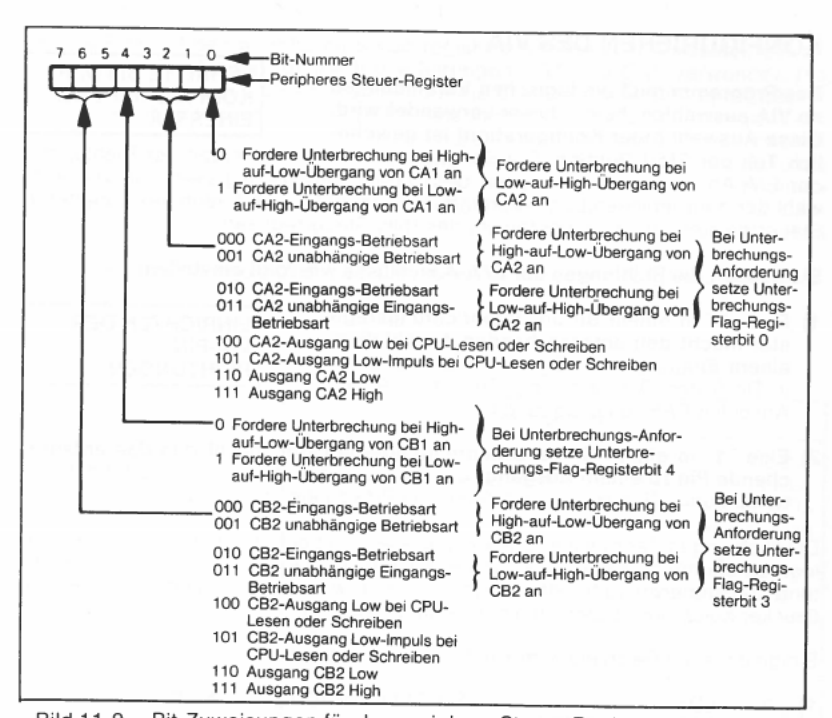


Bild 11-9. Bit-Zuweisungen für das periphere Steuer-Register des VIA 6522.

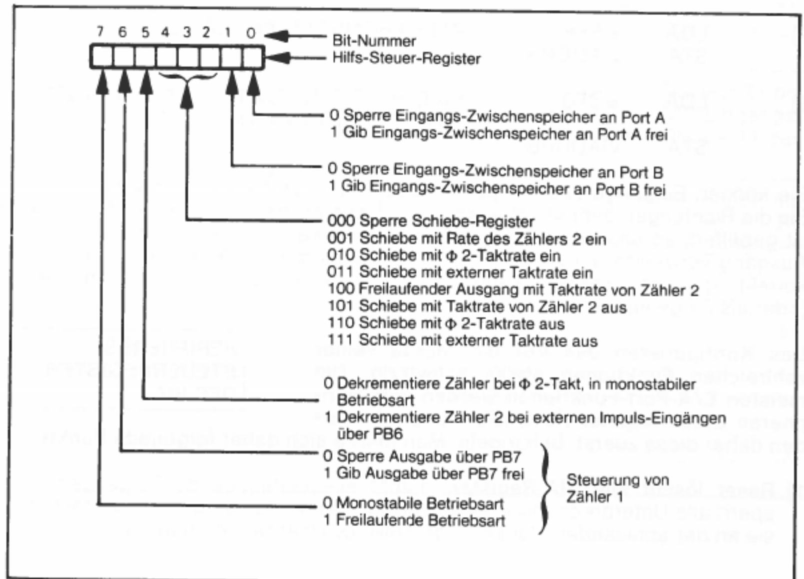


Bild 11-10. Bit-Zuweisungen für das Hilfs-Steuer-Register des VIA 6522.

KONFIGURIEREN DES VIA

SCHRITTE BEI DER KONFIGURIERUNG EINES VIA

Das Programm muß die logischen Verbindungen im VIA auswählen, bevor dieser verwendet wird. Diese Auswahl (oder Konfiguration) ist gewöhnlich Teil der Start-Routine. Die Schritte dienen zum Einrichten der Richtungen der E/A-Anschlüsse durch Laden des Datenrichtungs-Registers und zur Auswahl der erforderlichen Logik-Verbindungen im VIA durch Laden des peripheren Steuerregisters und falls erfrderlich, des Hilfs-Steuerregisters.

Sie können die Richtungen der E/A-Anschlüsse wie folgt einstellen:

EINRICHTEN DER VIA-PIN-RICHTUNGEN

1) **Eine "0" in einem Bit im Datenrichtungsregister macht den entsprechenden Anschluß zu einem Eingang.** Zum Beispiel macht eine "0" in Bit 5 des Datenrichtungs-Registers A den Anschluß PA5 zu einem Eingang.

2) **Eine "1" in einem Bit im Datenrichtungs-Register macht das das entsprechende Pin zu einem Ausgang.** Zum Beispiel macht eine "1" im Bit 3 des Datenrichtungs-Registers B den Anschluß PB3 zu einem Ausgang.

Die Richtungen der meisten E/A-Anschlüsse sind nach der Initialisierung festgelegt, da die meisten Eingangs- und Ausgangsleitungen Daten nur in einer Richtung transferieren (d.h., der Mikroprozessor wird niemals Daten von einem Drucker holen oder Daten zu einer Tastatur senden).

Einige einfache Beispiele zum Einstellen der Richtungen sind:

```
1)      LDA   #0        ;ALLE LEITUNGEN EINGÄNGE
        STA   VIADDRA

2)      LDA   #$FF      ;ALLE LEITUNGEN AUSGÄNGE
        STA   VIADDRB

3)      LDA   #$F0      ;MACHE LEITUNGEN 4-7 ZU AUSGÄNGEN,
                        ;  0-3 ZU EINGÄNGEN
        STA   VIADDRB
```

Sie können Eingänge und Ausgänge an einem einzelnen Port mischen, indem Sie die Richtungen der individuellen Anschlüsse entsprechend einrichten. Port B ist gepuffert, so daß sein Inhalt korrekt gelesen werden kann, auch wenn er als Ausgang verwendet wird. Port A ist nicht gepuffert, so daß sein Inhalt nur dann korrekt gelesen werden kann, wenn er nicht zu sehr belastet wird (oder als Eingänge dient).

PERIPHERES STEUERREGISTER DES VIA

Das Konfigurieren des VIA ist infolge seiner zahlreichen Funktionen etwas schwierig. Die meisten E/A-Port-Funktionen werden vom peripheren Steuerregister kontrolliert und wir werden daher diese zuerst behandeln. Man merke sich daher folgendde Punkte:

1) **Reset löscht alle VIA-Register,** macht alle Leitungen zu Eingängen und sperrt alle Unterbrechungen. Alle Flanken-Detektoren sind so eingestellt, daß sie an der abfallenden Flanke (High-auf-Low-Übergänge) triggern.

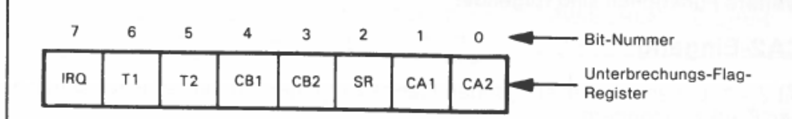
2) **Die Bits 0-3 des peripheren Steuerregisters werden zum Einrichten der logischen Verbindung für die Steuerleitungen CA1 und CA2 verwendet. Die Bits 4-7 haben den gleichen Zweck für die Stuerleitungen CB1 und CB2.**
3) **Die Steuerleitungen CA1 und CB1 sind immer Eingänge.** Die einzige Wahlmöglichkeit besteht darin, ob die entsprechenden Status-Zwischenspeicher (Unterbrechungsflag-Registerbits 1 und 4 – siehe Bild 11-11) auf abfallende Flanken (High-auf-Low-oder Negativ-Übergänge) oder auf ansteigende Flanken (Low-auf-High, oder positive Übergänge) eingestellt sind. Für CA1, Bit 0 = 0 für abfallende Flanken und 1 für ansteigende Flanken. Für CB1, Bit 4 = 0 für abfallende Flanken und 1 für ansteigende Flanken.
4) **Die Steuerleitungen CA2 und CB2 können entweder Eingänge oder Ausgänge sein** (siehe Tabelle 11-8 und 11-9). Für CA2, Bit 3 = 1 macht sie zu einem Ausgang und 0 zu einem Eingang.

Tabelle 11-8. Konfigurationen der Steuer-Leitung CA2 des VIA 6522.

PCR7	PCR6	PCR5	Betriebsart
0	0	0	Unterbrechungs-Eingangs-Betriebsart – Setze CB2-Unterbrechungs-Flag (IFR3) bei einem negativen Übergang des CB2-Eingangs-Signals. Lösche IFR3 beim Lesen oder Schreiben des peripheren B-Ausgangs-Registers.
0	0	1	Unabhängige Unterbrechungs-Eingangs-Betriebsart – Setze IFR3 bei einem negativen Übergang des CB2-Eingangs-Signals. Lesen oder Schreiben von ORB löscht das Unterbrechungs-Flag nicht.
0	1	0	Eingangs-Betriebsart – Setze CB2-Unterbrechungs-Flag bei einem positiven Übergang des CB2-Eingangs-Signals. Lösche das CB2-Unterbrechungs-Flag bei einem Lesen oder Schreiben von ORB.
0	1	1	Unabhängige Eingangs-Betriebsart – Setze IFR3 bei einem positiven Übergang des CB2-Eingangs-Signals. Lesen oder Schreiben von ORB löscht das CB2-Unterbrechungs-Flag nicht.
1	0	0	Quittierungs-Ausgangs-Betriebsart – Setze CB2 auf Low bei einer ORB-Schreib-Operation. Setze CB2 auf High zurück mit einem aktiven Übergang des CB1-Eingangs-Signals.
1	0	1	Impuls-Ausgangs-Betriebsart – Setze CB2 auf Low für die Dauer eines Zyklus nach einer ORB-Schreib-Operation.
1	1	0	Manuelle Ausgangs-Betriebsart – Der CB2-Ausgang wird in dieser Betriebsart auf Low gehalten.
1	1	1	Manuelle Ausgangs-Betriebsart – Der CB2-Ausgang wird in dieser Betriebsart auf High gehalten.

Tabelle 11-9. Konfigurationen der Steuer-Leitung CB2 des VIA 6522.

PCR3	PCR2	PCR1	Betriebsart
0	0	0	Eingangs-Betriebsart – Setze CA2-Unterbrechungs-Flag (IFR0) bei einem negativen Übergang des Eingangs-Signals. Lösche IFR0 bei einem Lesen oder Schreiben des peripheren A-Ausgangs-Registers.
0	0	1	Unabhängige Unterbrechungs-Eingangs-Betriebsart – Setze IFR0 bei einem negativen Übergang des CA2-Eingangs-Signals. Lesen oder Schreiben von ORA löscht das CA2-Unterbrechungs-Flag nicht.
0	1	0	Eingangs-Betriebsart – Setze CA2-Unterbrechungs-Flag bei einem positiven Übergang des CA2-Eingangs-Signals. Lösche IFR0 mit einem Lesen oder Schreiben des peripheren A-Ausgangs-Registers.
0	1	1	Unabhängige Unterbrechungs-Eingangs-Betriebsart – Setze IFR0 bei einem positiven Übergang des CA2-Eingangs-Signals. Lesen oder Schreiben von ORA löscht das CA2-Unterbrechungs-Flag nicht.
1	0	0	Quittierungs-Ausgangs-Betriebsart – Setze CA2-Ausgang auf Low bei einem Lesen oder Schreiben des peripheren A-Ausgangs-Registers. Lösche CA2 auf High mit einem aktiven Übergang an CA1.
1	0	1	Impuls-Ausgangs-Betriebsart – CA2 geht auf Low für die Dauer eines Zyklus nach einem Lesen oder Schreiben des peripheren A-Ausgangs-Registers.
1	1	0	Manuelle Ausgangs-Betriebsart – Der CA2-Ausgang wird in dieser Betriebsart auf Low gehalten.
1	1	1	Manuelle Ausgangs-Betriebsart – Der CA2-Ausgang wird in dieser Betriebsart auf High gehalten.



Bit-Nr.	Gesetzt durch	Gelöscht durch
0	Aktiver Übergang des Signals am Pin CA2	Lesen oder Schreiben des A-Port-Ausgangs-Registers (ORA) unter Verwendung der Adresse 0001.
1	Aktiver Übergang des Signals am Pin CA1	Lesen oder Schreiben des A-Port-Ausgangs-Registers (ORA) unter Verwendung der Adresse 0001.
2	Ausführung von acht Verschiebungen	Lesen oder Schreiben des Schiebe-Registers.
3	Aktiver Übergang des Signals am Pin CB2.	Lesen oder Schreiben des B-Port-Ausgangs-Registers.
4	Aktiver Übergang des Signals am Pin CB1.	Lesen oder Schreiben des B-Port-Ausgangs-Registers.
5	Ablauf von Zeitgeber 2.	Lesen des niederwertigen Zählers von T2 oder Schreiben des hochwertigen Zählers von T1.
6	Ablauf von Zeitgeber 1.	Lesen des niederwertigen Zählers von T1 oder Schreiben des hochwertigen Zwischenspeichers von T1.
7	Aktive und freigegebene Unterbrechungs-Bedingung	Tätigkeit, die die Unterbrechungsbedingung löscht.

Die Bits 0, 1, 3 und 4 sind die E/A-Quittierungs-Signale. Bit 7 (IRQ) ist 1, wenn eine der Unterbrechungen sowohl aktiv wie freigegeben ist (siehe Kapitel 12).

Bild 11-11. Das 6522-VIA-Unterbrechungs-Flag-Register.

Weitere Funktionen sind folgende:

CA2-Eingang

Bit 2 = 1, um an einer ansteigenden Flanke zu triggern, 0, um an einer abfallenden Flanke zu triggern.

Bit 1 = 1, um das Unterbrechungsflag-Registerbit 0 (der CA2-Eingangsstatus-Zwischenspeicher) unabhängig von Operationen am E/A-Port A zu machen, 0 um dieses Bit durch Operationen am E/A-Port A zu löschen.

Die unabhängige Betriebsart ist sehr nützlich, wenn CA2 für Zwecke verwendet wird, (wie einer Echtzeit-Uhr), die vollständig unabhängig von Datentransfers über den E/A-Port sind. Die reguläre Betriebsart ist nützlich, wenn CA2 als ein Quittierungssignal verwendet wird, das gelöscht werden muß, um die nächste E/A-Operation vorzubereiten (siehe Bilder 11-5 und 11-6).

CA2-Ausgang

Bit 2 = 1 um CA2 zu einem Pegel zu machen, 0 um es zu einem Impuls zu machen.

Wenn CA2 ein Pegel ist, ist Bit 1 dessen Wert.

Wenn CA2 ein Impuls ist, Bit 1 gleich 0, um CA2 auf Low zu bringen, wenn die CPU Daten zu oder vom Port A transferiert und verbleibt Low, bis ein aktiver Übergang an CA1 auftritt. Bit 1 = 1 um CA2 Low für einen Taktzyklus, nachdem die CPU Daten zu oder vom Port A transferiert hat, zu bringen.

CB2 wird genauso gehandhabt (Verwendung der Bits 7,6 und 5 des peripheren Steuerregisters und Bit 3 des Unterbrechungsflag-Registers) mit Ausnahme, daß Impulse an CB2 nur erzeugt werden, nachdem die Daten in den Port B geschrieben wurden. Um einen Impuls nach dem Lesen von Daten zu erzeugen, müssen Sie ein "dummy write" verwenden, das heißt:

```
LDA   VIAORB     ;HOLE DATEN VON PORT B
STA   VIAORB     ;ERZEUGE TAST-IMPULS VON PORT B
```


EINGANGS-ZWISCHENSPEICHER DES VIA

Die einzige E/A-Port-Funktion, die vom Hilfs-Steuerregister abhängig ist (Bild 11-10) ist das Zwischenspeichern des Eingangssignals. Bit 0 (für Port A) oder Bit 1 (für Port B) muß gesetzt werden um die Eingangsdaten bei einem aktiven Übergang auf der Steuerleitung 1 zwischenzuspeichern (wie durch das periphere Steuerregister bestimmt wird). **Beachten Sie die folgenden Eigenschaften der Zwischenspeicher-Funktion:**

1) RESET sperrt die Eingangs-Zwischenspeicher. Der VIA 6522 arbeitet dann wie der PIA 6520, der keine Eingangs-Zwischenspeicherung besitzt.

2) Für Port A werden die zwischengespeicherten Daten immer die Daten an den peripheren Anschlüssen sein. Da Port A nicht gepuffert ist, können diese Daten nicht dieselben sein wie die Daten im Ausgangsregister, wenn der Port für Ausgabe verwendet wird.

3) Für Port B sind die zwischengespeicherten Daten entweder die Daten an den peripheren Anschlüssen (für jene Pins, die als Eingänge definiert sind) oder der Inhalt der Ausgangsregister (für jene Pins, die als Ausgänge definiert sind).

Einige einfache Beispiele für die Aktivierung der Eingangs-Zwischenspeicher sind:

```
LDA   #%00000001
STA   VIAACR       ;AKTIVIERE ZWISCHENSPEICHER AN
                   ;   PORT A

LDA   #%00000010
STA   VIAACR       ;AKTIVIERE ZWISCHENSPEICHER AN
                   ;   PORT B

LDA   #%00000011
STA   VIAACR       ;AKTIVIERE ZWISCHENSPEICHER AN DEN
                   ;  PORTS A UND B
```

Beachten Sie, daß die Ausgangsports des 6522 automatisch zwischengespeichert werden, so wie die Ausgangsports des 6520.

BEISPIELE FÜR VIA-KONFIGURATIONEN

1) **Ein einfacher Eingangsport ohne Steuerleitungen** (wie für einen Satz von Schaltern benötigt wird):

```
LDA   #0
STA   VIAPCR      ;MACHE ALLE STEUERLEITUNGEN ZU EIN-
                  ; GÄNGEN
STA   VIADDRA     ;MACHE PORT-A-LEITUNGEN ZU EIN-
                  ; GÄNGEN
```

Erinnern Sie sich daran, daß Reset alle internen Register löscht, so daß diese Sequenz nicht unbedingt erforderlich ist. Die gleiche Sequenz kann verwendet werden, wenn eine High-auf-Low-Flanke (abfallende Flanke) auf der Steuerleitung CA1 ein Data Ready oder Peripheral Ready anzeigt.

2) **Ein einfacher Ausgangsport ohne Steuerleitungen** (wie für einen Satz einzelner LED-Anzeigen benötigt wird):

```
LDA   #0
STA   VIAPCR      ;MACHE ALLE STEUERLEITUNGEN ZU EIN-
                  ; GÄNGEN
LDA   #$FF
STA   VIADDRB     ;MACHE PORT-B-LEITUNGEN ZU AUS-
                  ; GÄNGEN
```

3) **Ein Eingangsport ohne aktives Low-auf-High-Signal DATA READY, das CA1 zugeführt wird** (wie für eine nicht codierte Tastatur erforderlich ist):

```
LDA   #0
STA   VIADDRA     ;MACHE PORT-A-LEITUNGEN ZU EIN-
                  ; GÄNGEN
LDA   #1          ;MACHE ANSTEIGENDE FLANKE AKTIV
STA   VIAPCR
```

Bit 1 des peripheren Steuerregisters wird gesetzt, um Low-auf-High-Übergänge auf der Steuerleitung CA1 zu erkennen. Ein derartiger Übergang wird Bit 1 des Unterbrechungsflag-Registers (siehe Bild 11-10) setzen. Das Lesen von Daten vom Port wird dieses Bit löschen (siehe Tabelle bei Bild 11-11). Das Eingangs-Zwischenspeichern kann erfolgen, indem Bit 0 des Hilfs-Steuerregisters gesetzt wird.

4) **Ein Ausgangsport, der einen kurzen Impuls erzeugt, um DATA READY oder OUTPUT READY anzuzeigen** (dies könnte für das Multiplexen von Anzeigen oder für das Liefern eines Signales DATA AVAILABLE zu einem Drucker erforderlich sein):

```
LDA   #$FF
STA   VIADDRB     ;MACHE PORT-B-LEITUNGEN ZU AUS-
                  ; GÄNGEN
LDA   #%10100000
STA   VIAPCR
```

Der kurze Impuls auf der Steuerleitung CB2 wird nach jeder Ausgangs-Operation auftreten. Bit 7 des peripheren Steueregisters ist 1 um CB2 zu einem Ausgang zu machen, Bit 6 ist 0, um CB2 zu einem Impuls zu machen, und Bit 5 ist 1, um CB2 zu einem kurzen Impuls (ein Taktzyklus) zu machen, folgend nach jeder Ausgabe.

5) **Ein Eingangsport mit einem Quittierungs-Impuls Input Acknowledge (Eingangs-Bestätigung), der verwendet werden kann, um einen peripheren Baustein mitzuteilen, daß die vorhergehenden Daten angenommen wurden** (und das der Computer bereit für weitere ist):

```
LDA   #0
STA   VIADDRA     ;MACHE PORT-A-LEITUNGEN ZU EIN-
                  ; GÄNGEN
LDA   #%00001000  ;STEUERLEITUNG 2 = QUITTIERUNGSBE-
                  ; STÄTIGUNG
```

Der Impuls auf der Steuerleitung CA2 wird nach jeder Eingabe-oder Ausgabe-Operation auftreten. Er wird bis zum nächsten aktiven Übergang auf der Steuerleitung CA1 auf Low bleiben. Bit 3 des peripheren Steuerregisters ist 1, um CA2 zu einem Ausgang zu machen, Bit 2 ist 0, um CA2 zu einem Impuls zu machen, und Bit 1 ist 0, um CA2 zu einem Bestätigungssignal mit aktiv Low zu machen, das bis zum nächsten aktiven Übergang an CA1 dauert. Beachten Sie, daß der aktive Übergang an CA1 eine abfallende Flanke ist, da Bit 0 des peripheren Steuerregisters gleich 0 ist. Diese Konfiguration ist geeignet für zahlreiche Bildschirm-Terminals, die eine vollständige Quittierung benötigen.

6) **Ein Ausgangsport mit zwischengespeichertem Steuerbit aktiv-Low** (zwischengespeicherter Ausgang oder Pegel-Ausgang). Ein derartiges Ausgangsbit kann zum Ein- oder Ausschalten eines peripheren Gerätes oder zum Steuern seiner Betriebsart dienen.

```
LDA   #$FF        ;MACHE PORT-B-LEITUNGEN ZU AUS-
                  ; GÄNGEN
STA   VIADDRB
LDA   #%11000000  ;STEUERLEITUNG 2 = ZWISCHENGESPEI-
                  ; CHERTER NULLPEGEL
STA   VIAPCR
```

Bit 7 = 1, um die Steuerleitung CB2 zu einem Ausgang zu machen, Bit 6 = 1, um es zu einem Pegel zu oder einem gespeicherten Bit zu machen, und Bit 5 = 0, um den Pegel aktiv null zu machen. Dieses Bit wird nicht durch Operationen am E/A-Port oder Ausgangsregister beeinflußt. Sein Wert kann durch Ändern von Bit 5 des peripheren Steuerregisters geändert werden, d.h.,

```
LDA   VIAPCR
ORA   #%00100000  ;MACHE PEGEL EINS
STA   VIAPCR
LDA   VIAPCR
AND   #%11011111  ;MACHE PEGEL NULL
STA   VIAPCR
```

Sie können die Konfiguration zur Erzeugung eines Impulses mit aktiv High oder aktiv Low verwenden, oder zum Liefern von Impulsen mit software-gesteuerten Längen.

VERWENDUNG DES VIA ZUM TRANSFER VON DATEN

VIA-EINGABE/ AUSGABE

Sobald der VIA konfiguriert wurde, können Sie seine Datenregister wie jeden anderen Speicherplatz verwenden (so wie beim PIA). Der gebräuchliche Weg zum Transferieren von Daten, Status und Steuerung geschieht mit den Befehlen Load Accumulator, Store Accumulator, Bit Test und Compare. Beachten Sie, daß das Ausgangsregister A auf zwei Wegen adressiert werden kann – einer mit Quittierung (Adresse 1) und einer ohne Adressierung (Adresse F). Die Adresse ohne Quittierung gestattet Ihnen, CA1 unabhängig von dem mit dem E/A-Port A verbundenen peripheren Baustein zu verwenden. Diese Steuerleitung könnte für einen Alarm, Takt-Eingang, zur Steuerung eines Interface, oder als zusätzlicher Steuereingang von anderen peripheren Geräten verwendet werden. Das Unterbrechungsflag für diesen Eingang kann direkt gelöscht werden, indem die entsprechenden Bits im Unterbrechungsflag-Register gelöscht werden (siehe Bild 11-11). Die andere Adresse für das Ausgangsregister A und die unabhängigen Betriebsart für die Steuerleitungen CA2 und CB2 gestatten die Verwendung von Steuerleitungen, ohne sich über die automatischen Quittierungs-Eigenschaften des VIA kümmern zu müssen.

UNTERBRECHUNGSFLAG-REGISTER DES VIA

UNTERBRECHUNGS-FLAG-REGISTER DES VIA

Wir haben das VIA-Unterbrechungsflag-Register (siehe Bild 11-11) bei verschiedenen Gelegenheiten erwähnt. **Die Tabelle in Bild 11-11 erklärt die Bedeutung der verschiedenen Bits** (Bit 7 ist ein universelles Unterbrechungs-Anforderungsbit, das 1 ist, wenn irgendeine Unterbrechung sowohl aktiv wie freigegeben ist).

Jedes der Flags im Unterbrechungsflag-Register kann ausdrücklich durch Schreiben einer logischen 1 in die entsprechende Bit-Position gelöscht werden. Dieses Verfahren ist dann nützlich, wenn die Steuerleitungen unabhängig von den Datenports verwendet werden (wie bei der unabhängigen Eingabe-Betriebsart, beschrieben in den Tabellen 11-8 und 11-9), oder wenn als Reaktion auf das zu setzende Flag kein Datentransfer tatsächlich erforderlich ist. Einige Beispiele für ein ausdrückliches Löschen der Flags sind:

```
        LDA   #%00000010
        STA   VIAIFR       ;LÖSCHE CA1-UNTERBRECHUNGSFLAG
        LDA   #%00001000
        STA   VIAIFR       ;LÖSCHE CB2-UNTERBRECHUNGSFLAG
        LDA   #%11111111
        STA   VIAIFR       ;LÖSCHE ALLE UNTERBRECHUNGSFLAGS
```

Der in Bit 7 geschriebene Wert ist ohne Bedeutung, da dieses Flag nicht ausdrücklich von der CPU gesetzt oder gelöscht werden kann.

Die Bits 0, 1, 3 und 4 des VIA-Unterbrechungsflag-Registers dienen häufig als Quittierungs-Statusbit, wie etwa Data Ready oder Peripheral Ready. Sie können ihre Werte durch entsprechendes Maskieren oder Schiebe-Operationen prüfen.

```
        LDA   VIAIFR
        AND   #%00000010   ;IST CA1-FLAG GESETZT?
        BNE   DEVRDY       ;JA, BAUSTEIN BEREIT
        LDA   VIAIFR
        AND   #%00010000   ;IST CB1-FLAG GESETZT?
        BNE   DEVRDY       ;JA, BAUSTEIN BEREIT
```

Das Flag wird automatisch durch Lesen oder Schreiben des entsprechenden Ports oder durch spezielles Löschen des Bits im Unterbrechungsflag-Register gelöscht. Das folgende Programm wird auf ein Ready-Flag warten, das dem Eingang CA1 für ein High zugeordnet ist:

```
WAITR   LDA   VIAIFR
        AND   #%00000010   ;IST CA1-FLAG GESETZT?
        BEQ   WAITR        ;NEIN, WARTE
```

Wie könnten Sie dieses Programm ändern, um die Ready-Leitungen zu handhaben, die zu CA2, CB1 oder CB2 geführt sind?

Beachten Sie, daß die Flags gesetzt bleiben, bis irgendeine Operation sie löscht. Wenn keine Operation tatsächlich erforderlich ist, kann irgendeine "Blind" (dummy)-Operation (wie das Lesen des Ports und Löschen der Daten) erforderlich sein, um einfach das Flag zu löschen. Seien Sie besonders sorgfältig in Fällen, bei denen die CPU nicht bereit für Daten ist oder keine Ausgangsdaten zu senden hat. Offensichtlich ist in Fällen, bei denen der Zweck der Operationen nicht sofort deutlich wird, eine sorgfältige Dokumentation sehr wesentlich.

VIA-ZEITGEBER[9,10]

VIA-ZEITGEBER

Wie wir früher beschrieben haben, enthält der VIA zwei 16-Bit-Zähler/Zeitgeber. Diese Zeitgeber werden wie folgt gehandhabt:

1) **Sie können wie sechs Speicherplätze gelesen oder geschrieben werden, vier für Zeitgeber 1 und zwei für Zeitgeber 2** (siehe Tabelle 11-7).

2) **Ihre Betriebsarten werden durch die Bits 5, 6 und 7 des Hilfs-Steuerregisters bestimmt** (siehe Bild 11-10).

3) **Ihr Status kann durch Prüfen der Bits 5 und 6 des Unterbrechungsflags-Registers bestimmt werden** (siehe Bilkd 11-11).

Die Zeitgeber können wie folgt verwendet werden:

1) **Zur Erzeugung eines einzelnen Zeitintervalls.** Der Zeitgeber muß mit der Anzahl der erforderlichen Taktimpulse geladen werden.

2) **Zum Zählen von Impulsen am Anschluß PB6** (nur Zeitgeber 2). Der Zeitgeber muß mit der Anzahl der zu zählenden Impulse geladen werden. Diese Verwendung von PB6 nimmt den Vorrang über seine normale Verwendung als E/A-Anschluß ein.

3) **Zur Erzeugung kontinuierlicher Zeitintervalle (nur Zeitgeber 1), zur Verwendung in Echtzeit-Anwendungen.** Der Zeitgber muß mit der Anzahl der Taktimpulse pro Intervall geladen werden.

4) **Zur Erzeugung eines einzelnen Impulses oder einer aufeinanderfolgenden Serie von Impulsen am Anschluß PB7** (nur Zeitgeber 1). Der Zeitgeber muß mit der Anzahl der Taktimpulse pro Intervall geladen werden. Diese Verwendung von PB7 übernimmt den Vorrang über seine normale Verwendung als ein E/A-Anschluß.

ARBEITSWEISE DES ZEITGEBERS 2 IM VIA 6522

Zeitgeber 2 ist einfacher als Zeitgeber 1 und kann nur zur Erzeugung eines einzelnen Zeitintervalls verwendet werden (die monostabile Betriebsart) **oder zur Zählung von Impulsen an Anschluß PB6.** Bit 5 des Hilfs-Steuerregisters wählt die Betriebsart aus:

Bit 5 = 0 für monostabilen Betrieb, 1 für das Zählen von Impulsen.

Der 16-Bit-Zeitgeber belegt zwei Speicherplätze (siehe Tabelle 11-7). Die erste Adresse wird zum Lesen oder Schreiben der acht höchstwertigen Bits verwendet. Das Lesen dieser Adresse löscht auch das Unterbrechungsflag des Zeitgebers 2 (Bild 11-11). Die zweite Adresse wird zum Lesen oder Schreiben der acht höchstwertigen Bits verwendet. Das Schreiben in diese Adresse lädt die Zähler, löscht das Unterbrechungsflag des Zeitgebers 2 und startet die zeitliche Operation. Der Abschluß dieser Operation setzt das Unterbrechungsflag des Zeitgebers 2 (Bit 5 des Unterbrechungsflag-Registers, wie in Bild 11-11 gezeigt).

Beispiele für die Arbeitsweise des Zeitgebers 2 wären folgende:

1) Warten 1024 (0400_{16}) Taktimpulse.

```
        LDA  #0          ;BRINGE ZEITGEBER 2 IN MONOSTABILE
                         ;  BETRIEBSART (BIT 5 = 0)
        STA  VIAACR
        STA  VIAT2L      ;MACHE IMPULSELÄNGE 0400HEX
        LDA  #4
        STA  VIAT2H      ;STARTE ZEITGEBER-INTERVALL
        LDA  #%00100000  ;HOLE MASKE FÜR UNTERBRECHUNGS-
                         ;  FLAG VON ZEITGEBER 2
WAITD   BIT  VIAIFR      ;IST FLAG VON ZEITGEBER 2 GESETZT?
        BEQ  WAITD       ;NEIN, INTERVALL NICHT BEENDET
        LDA  VIAT2L      ;JA, LÖSCHE UNTERBRECHUNGSFLAG
        BRK
```

Beachten Sie folgende Schritte in diesem Programm:

a) Versetzen des Zeitgebers in die monostabile Betriebsart durch Löschen von Bit 5 des Hilfs-Steuerregisters.

b) Laden des Zeitgebers mit der Anfangszählung (0400_{16}), erforderlich, um das richtige Intervall zu ergeben. Das Laden des MSBs des Zeitgebers startet auch die eigentliche Operation.

c) Warten bis das Intervall abgeschlossen ist. Nach Ablauf der Zeit wird Bit 5 des Unterbrechungsflag-Registers gesetzt.

d) Löschen des Unterbrechungsflags, so daß es nicht andere Operationen stört. Der Befehl LDA VIAT2L führt diese Funktion aus.

2) Erzeugen einer Verzögerung, deren Länge durch 10 Impulse am Anschluß PB6 gegeben ist.

```
        LDA   #0
        STA   VIADDRB      ;MACHE PORT B ZU EINGÄNGEN
        LDA   #%00100000   ;BRINGE ZEITGEBER 2 IN IMPULSZÄHL-
                           ;  BETRIEBSART (BIT 5 = 1)
        STA   VIAACR
        LDA   #10          ;MACHE IMPULSZÄHLUNG 10
        STA   VIAT2L
        LDA   #0
        STA   VIAT2H       ;STARTE IMPULSZÄHLUNG
        LDA   #%00100000   ;HOLE MASKE FÜR UNTERBRECHUNGS-
                           ;  FLAG VON ZEITGEBER 2
WAITD   BIT   VIAIFR       ;IST FLAG VON ZEITGEBER 2 GESETZT?
        BEQ   WAITC        ;NEIN, ZÄHLUNG NICHT BEENDET
        LDA   VIAT2L       ;JA, LÖSCHE UNTERBRECHUNGSFLAG
        BRK
```

Dieses Programm ist das gleiche wie das vorausgehende Beispiel, mit Ausnahme, daß die Betriebsart von Zeitgeber 2 unterschiedlich ist. Hier könnte das Eingangssignal am Anschluß PB6 ein periodischer Takt oder eine Leitung sein, die einfach bei jedem Auftreten irgendeiner externen Operation gepulst wird.

ARBEITSWEISE DES ZEITGEBERS 1 IM VIA 6522

Der Zeitgeber 1 hat vier Betriebsarten (siehe Bild 11-10), **die die Erzeugung eines einzelnen Zeitintervalls** (monostabile Betriebsart) **oder einer aufeinander folgenden Serie von Intervallen** (freilaufende Betriebsart) **gestatten.** Ferner kann jede Lade-Operation einen Ausgangsimpuls an PB7 erzeugen, der zum Steuern externer Hardware verwendet werden kann. Die Bits 6 und 7 des Hilfs-Steuerregisters bestimmen die Betriebsart des Zeitgebers 2 wie folgt:

Bit 7 = 1 zum Erzeugen von Ausgangsimpulsen an Anschluß PB7, 0 zum Sperren derartiger Impulse (in der freilaufenden Betriebsart wird PB7 jedesmal, wenn der Zähler Null erreicht invertiert).

Bit 6 = 1 für freilaufende Betriebsart, 0 für monostabile Betriebsart.

Der Zeitgeber 1 belegt vier Speicheradressen (siehe Tabelle 11-7). Die ersten beiden Adressen werden zum Lesen oder Schreiben der Zähler verwendet. Das Schreiben in die zweite Adresse lädt die Zähler, löscht das Unterbrechungsflag des Zeitgebers 1, und startet die zeitliche Operation. Die nächsten zwei Adressen werden zum Lesen oder Schreiben in die Speicher ohne Beeinflussung der Zähler verwendet. Dies gestattet die Erzeugung komplexer Spannungen in der freilaufenden Betriebsart. Das Schreiben in die höchstwertigen Bits der Zwischenspeicher löscht auch das Unterbrechungsflag von Zeitgeber 1.

Beispiele für die Arbeitsweise des Zeitgebers 1 wären folgende:

1) Warte auf das Ablaufen von 4096 (1000_{16}) Taktimpulsen, vor der Erzeugung eines Ausgangssignals am Anschluß PB7.

```
        LDA   #0           ;BRINGE ZEITGEBER 1 IN MONOSTABILE
                           ;  BETRIEBSART, KEINE AUSGABE-
                           ;  BETRIEBSART
        STA   VIAACR
        STA   VIAT1L       ;IMPULSLÄNGE = 1000 HEX
        LDA   #$10
        STA   VIAT1CH      ;STARTE ZEITGEBER-INTERVALL
        LDA   #%01000000   ;HOLE MASKE FÜR UNTERBRECHUNGS-
                           ;  FLAG VON ZEITGEBER 1
WAITD   BIT   VIAIFR       ;IST FLAG VON ZEITGEBER 1 GESETZT?
        BEQ   WAITD        ;NEIN, INTERVALL NICHT ABGESCHLOSSEN
        LDA   VIAT1L       ;JA, LÖSCHE UNTERBRECHUNGSFLAG VON
                           ;  ZEITGEBER 1
        BRK
```

Die einzigen Änderungen gegenüber dem Programm für Zeitgeber 2 sind die unterschiedlich verwendeten Adressen zum Laden der Impulslänge und die unterschiedliche Bit-Position (Bit 6 anstatt von Bit 5) für die Prüfung des Unterbrechungsflags.

2) Erzeuge eine Unterbrechung alle 2048 (0800_{16}) Taktimpulse und erzeuge eine aufeinanderfolgende Serie von Zyklen am Anschluß PB7 mit einer halben Breite von 1024 Taktimpulsen.

```
    LDA   #$FF         ;MACHE PORT-B-LEITUNGEN ZU AUS-
                       ;  GÄNGEN
    STA   VIADDRB
    LDA   #%11000000   ;BRINGE ZEITGEBER 1 IN KONTINUIERLICHE
                       ;  BETRIEBSART MIT AUSGABE ZU PB7
    STA   VIAACR
    LDA   #0           ;MACHE IMPULSLÄNGE 0800 HEX
    STA   VIAT1L
    LDA   #8
    STA   VIAT1CH      ;STARTE ZEITGEBER-INTERVALLE
    BRK
```

Diese Routine wird eine aufeinanderfolgende Serie von Intervallen erzeugen, die durch das Setzen des Unterbrechungsflags des Zeitgebers 1 markiert sein werden (Bit 6 des Unterbrechungsflag-Registers). Das Hauptprogramm kann auf das Auftreten jeder Unterbrechung (mit der Warte-Routine von Beispiel 1) Ausschau halten, oder (empfindlicher) das Ende jedes Zeitintervalles kann eine Unterbrechung erzeugen (siehe Kapitel 12). Der Pegel an PB7 wird am Ende jedes Zeitintervalles invertiert werden (er wird Low werden, wenn das erste Intervall startet). Zeitgeber 1 wird kontinuierlich mit Werten in den Zwischenspeicher laufen, die automatisch zurück in den Zähler geladen werden, jedesmal wenn die Zähler Null erreichen.

DIE MEHRFUNKTIONS-BAUSTEINE 6530 UND 6532

6530 UND 6532 MEHRFUNKTIONS-BAUSTEINE

Die Bausteine 6530 und 6532 enthalten Speicher sowie E/A-Ports. Sie werden manchmal als Kombinations-Chips, Mehrfunktions-Hilfsbausteine oder TIMER-Chips (RIOTs) bezeichnet.
Der Baustein 6530 beinhaltet:

- 1024 Bytes ROM
- 64 Bytes RAM
- Zwei 8-Bit-E/A-Ports (A und B) obwohl die Pins 5 bis 7 von Port B häufig für Chip-Auswahl und Unterbrechungs-Ausgabe verwendet werden.
- Ein 8-Bit-Zeitgeber

Bild 11-12 ist ein Blockschaltbild des Bausteins 6530 und Tabelle 11-10 beschreibt seine interne Adressierung. Der Baustein 6532 hat:

- 128 Bytes RAM
- Zwei 8-Bit-E/A-Ports (A und B) obwohl Pin 7 von Port A häufig als ein Tasteingang verwendet wird, vergleichbar mit den Anschlüssen CA1 oder CB1 eines Bausteines 6520 oder 6522.
- Ein 8-Bit-Zeitgeber

Bild 11-13 ist ein Blockschaltbild des Bausteins 6532 und Tabelle 11-11 beschreibt seine interne Adressierung. Beachten Sie, daß Bausteine 6532 kein ROM enthalten.

Die folgenden Eigenschaften der Bausteine 6530 und 6532 sollten beachtet werden:

1) Sie enthalten weder irgendwelche bestimmten Steuerleitungen, obwohl Pin 7 von Port A eines Bausteines 6532 für diesen Zweck verwendet werden kann.

2) Beide enthalten einen einzelnen 8-Bit-Zeitgeber mit einem Vorteiler, der es gestattet, Zeit-Intervalle mit Faktoren von 1, 8, 64 oder 1024 Taktimpulsen zu multiplizieren. Der Zeitgeber kann daher zur Lieferung von Zeitintervallen verwendet werden, die wesentlich länger als die grundlegende 256 Taktzahl sind.

3) Das Ende des Zeitintervalls bewirkt entweder eine Unterbrechung oder setzt ein Flag, das gelesen werden kann.

Die Bausteine 6530 und 6532 werden in den bekannten Ein-Platinen-Mikrocomputer wie KIM, VIM, SYM und AIM-65 verwendet.

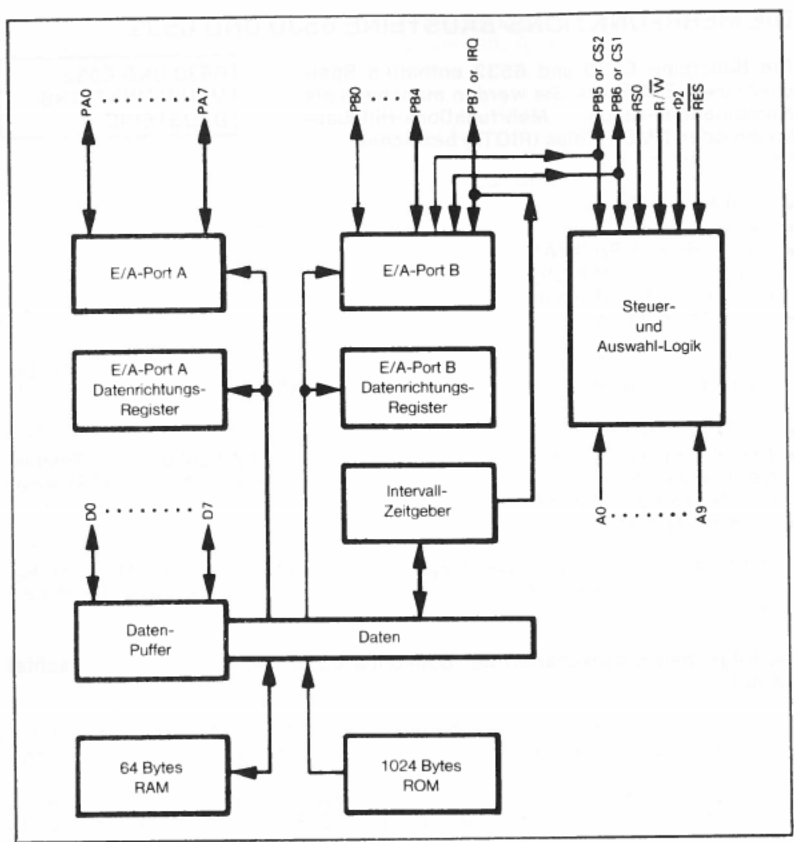


Bild 11-12. Blockschaltbild des Mehrfunktions-Bausteins 6530.

Tabelle 11-10. Interne Adressierung des Mehrfunktions-Bausteins 6530.

Primäre Auswahl			Adressierte Speicherplätze				
RSO	RAM-Auswahl*	E/A-Zeitgeber-Auswahl*					
1	X	X	A0 – A9 adressiert direkt eines der 1024 ROM-Bytes				
0	1	0	A0 – A5 adressiert direkt eines der 64 RAM-Bytes				
			Sekundäre Auswahl				Interpretation
			A3	A2	A1	A0	
0	0	1	X	0	0	0	Zugriff auf E/A-Port A
0	0	1	X	0	0	1	Zugriff auf Datenrichtungs-Register von E/A-Port A
0	0	1	X	0	1	0	Zugriff auf E/A-Port B
0	0	1	X	0	1	1	Zugriff auf Datenrichtungs-Register von E/A-Port B
0	0	1W	0	1	X	X	Sperre IRQ
0	0	1W	1	1	X	X	Gib IRQ frei
0	0	1W	X	1	0	0	Schreibe zu Zeitgeber, dann dekrem. alle ϕ2-Impulse
0	0	1W	X	1	0	1	Schreibe zu Zeitgeb., dann dekrem. alle 8 ϕ2-Imp.
0	0	1W	X	1	1	0	Schreibe zu Zeitgeb., dann dekrem. alle 64 ϕ2-Imp.
0	0	1W	X	1	1	1	Schreibe zu Zeitgeb., dann dekrem. alle 1024 ϕ2-Imp.
0	0	1R	X	1	X	0	Lies Zeitgeber
0	0	1R	X	1	X	1	Lies Unterbrechungs-Flags

*RAM-Auswahl und E/A-Auswahl sind "wahr" wenn 1 oder "falsch wenn 0. Wahr und falsch sind Funktionen Ihrer Spezifikation. Sie können die Kombination von Adressen-Leitungen spezifizieren, die eine Leitungs-Bedingung mit "wahr" erzeugen.

X bedeutet "beliebig. Bits können 0 oder 1 sein.
1R bedeutet Auswahl während eines Lesens.
1W bedeutet Auswahl während eines Schreibens.

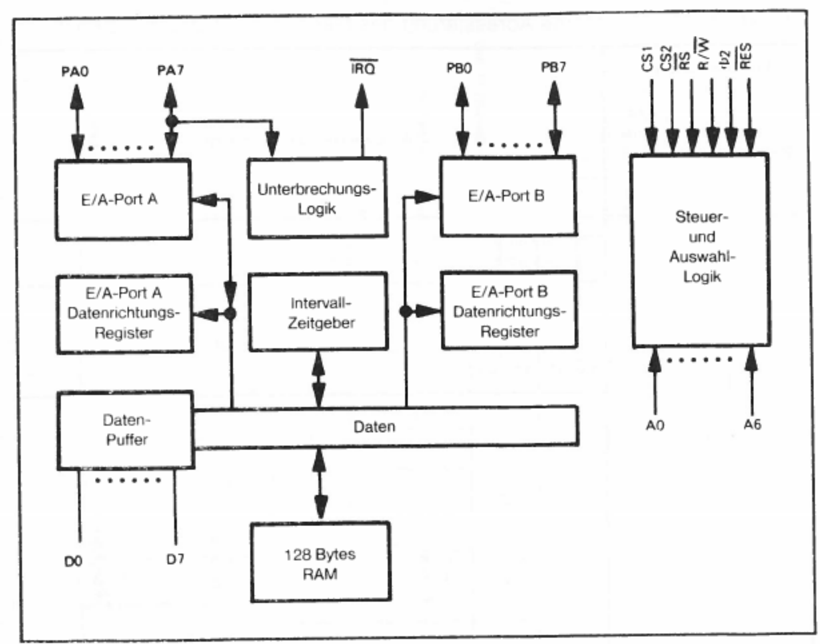


Bild 11-13. Blockschaltbild des Mehrfunktions-Bausteins 6532.

Tabelle 11-11. Interne Adressierung des Mehrfunktions-Bausteins 6532.

Primäre Auswahl		Sekundäre Auswahl					Interpretation
RAM-Auswahl	E/A-Zeitgeber-Auswahl	A4	A3	A2	A1	A0	
1	0	X	X	X	X	X	A0 – A6 adressiert direkt eines der 128 RAM-Bytes
0	1	X	X	0	0	0	Zugriff auf E/A-Port A
0	1	X	X	0	0	1	Zugriff auf Datenrichtungs-Register des E/A-Ports A
0	1	X	X	0	1	0	Zugriff auf E/A-Port B
0	1	X	X	0	1	1	Zugriff auf Datenrichtungs-Register des E/A-Ports B
0	1W	1	0	1	X	X	Sperre $\overline{IRQ}$
0	1W	1	1	1	X	X	Gib $\overline{IRQ}$ frei
0	1W	1	X	1	0	0	Schreibe zu Zeitgeber, dekrementiere alle ϕ2-Impulse
0	1W	1	X	1	0	1	Schreibe zu Zeitgeber, dekrem. alle 8 ϕ2-Imp.
0	1W	1	X	1	1	0	Schreibe zu Zeitgeber, dekrem. alle 64 ϕ2-Imp.
0	1W	1	X	1	1	1	Schreibe zu Zeitgeber, dekrem. alle 1024 ϕ2-Imp.
0	1R	X	X	1	X	0	Lies Zeitgeber
0	1R	X	X	1	X	1	Lies Unterbrechungs-Flags
0	1W	0	X	1	X	0	Fordere Unterbrech. bei High-auf-Low-Überg. v. PA7 an
0	1W	0	X	1	X	1	Fordere Unterbrech. bei Low-auf-High-Überg. v. PA7 an
0	1W	0	X	1	0	X	Gib PA7-Unterbrechungs-Anforderung frei
0	1W	0	X	1	1	X	Sperre PA7-Unterbrechungs-Anforderung

X bedeutet "beliebig". Bits können 0 oder 1 sein.
1R bedeutet Lese-Zugriff. 1W bedeutet Schreib-Zugriff.

BEISPIELE

Eine Taste

Zweck: Die Anpassung einer einzelnen Taste an einen Mikroprozessor 6502 mittels eines VIA 6522. Die Taste ist ein mechanischer Schalter, der (einen Logikpegel "0") durch einen geschlossenen Kontakt liefert, wenn die Taste betätigt wird.

Bild 11-14 zeigt die erforderliche Schaltung für die Anpassung der Taste. Sie verwendet ein Bit eines PIA 6522, der als Puffer arbeitet. Es ist keine Zwischenspeicherung erforderlich, da die Taste für mehrere CPU-Zyklen geschlossen bleibt. Das Drücken der Taste legt das Eingangs-Bit des VIA an Masse. Der Pull-up-Widerstand sichert, daß das Eingangs-Bit "1" ist, wenn die Taste nicht gedrückt ist.

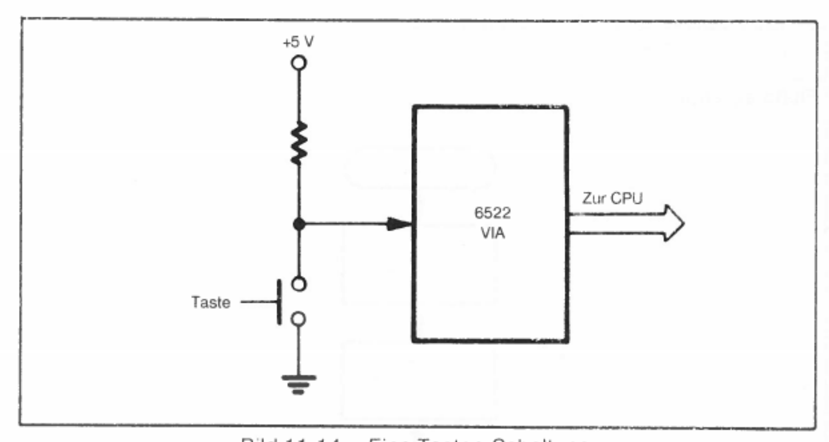


Bild 11-14. Eine Tasten-Schaltung

Programmbeispiele:

Wir wollen zwei Aufgaben mit dieser Schaltung ausführen. Diese sind:

1) Setze einen Speicherplatz, basierend auf den Zustand der Taste.

2) Zähle, wie oft die Taste gedrückt wurde.

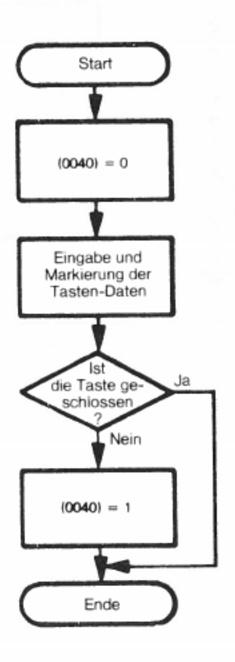
Aufgabe 1: Setze Speicherplatz 0040 auf 1, wenn die Taste nicht gedrückt wird, und auf 0, wenn sie gedrückt wird.

Beispieie:

1) Taste offen (d.h., nicht gedrückt)
   Ergebnis: (0040) = 01

2) Taste geschlossen (d.h., gedrückt)
   Ergebnis (0040) = 00

Flußdiagramm:



Quellprogramm:

```
        LDA   #0
        STA   VIAPCR     ;MACHE ALLE STEUERLEITUNGEN ZU EIN-
                         ;  GÄNGEN
        STA   VIADDRA    ;MACHE PORT-A-LEITUNGEN ZU EIN-
                         ;  GÄNGEN
        STA   $40        ;MARKE = 0
        LDA   VIAORA     ;LIES TASTEN-POSITION
        AND   #MASK      ;IST DIE TASTE GESCHLOSSEN (LOGISCH
                         ;  NULL?)
        BEQ   DONE       ;JA, ZU DONE
        INC   $40        ;NEIN, MARKE = 1
DONE    BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
0000	A9		LDA	#0
0001	00			
0002	8D		STA	VIAPCR
0003 } 0004 }	VIAPCR			
0005	8D		STA	VIADDRA
0006 } 0007 }	VIADDRA			
0008	85		STA	$40
0009	40			
000A	AD		LDA	VIAORA
000B } 000C }	VIAORA			
000D	29		AND	#MASK
000E	MASK			
000F	F0		BEQ	DONE
0010	02			
0011	E6		INC	$40
0012	40			
0013	00	DONE	BRK	

Die Adressen VIAPCR (Peripheral Control Register), VIADDRA (Data Direction Register A), und VIAORA (Output Register A) hängen davon ab, wie der VIA mit Ihrem Mikrocomputer verbunden ist. Die VIA-Steuerleitungen werden in diesem Beispiel nicht verwendet. Der Inhalt des peripheren Steuerregisters ist daher irrelevant, wir haben jedoch dieses Register vorsichtshalber gelöscht. Wir haben angenommen (wie dies gewöhnlich der Fall ist), daß die VIA-Adressen nicht auf Seite Null liegen.

MASK hängt von dem Bit ab, mit dem die Taste verbunden ist. Es besitzt eine Eins in der Tastenposition und sonst Nullen.

Tasten-Position (Bit-Nummer)	Maske	
	Binär	Hexadezimal
0	00000001	01
1	00000010	02
2	00000100	04
3	00001000	08
4	00010000	10
5	00100000	20
6	01000000	40
7	10000000	80

Wenn die Taste mit Bit 6 oder Bit 7 des VIA-Eingangsport verbunden ist, kann das Programm einen Bit-Test-Befehl zu Setzen des Überlauf- oder Vorzeichen-Bits verwenden und hierbei den Zustand der Taste bestimmen. Zum Beispiel:

Bit 7

```
BIT     VIAORA   ;IST TASTE GESCHLOSSEN (LOGISCH
                 ;  NULL)?
BPL     DONE     ;JA, DONE
```

Bit 6

```
BIT     VIAORA   ;IST TASTE GESCHLOSSEN (LOGISCH
                 ;  NULL)?
BVC     DONE     ;JA, DONE
```

Beachten Sie die Verwendung von BVC oder BVS zum Prüfen des Wertes von Bit 6.

Wir könnten auch Schiebebefehle verwenden, wenn die Taste mit den Bits 0, 6 oder 7 verbunden ist. Die Sequenz für Bit 0 lautet:

```
        LSR     VIAORA   ;IST TASTE GESCHLOSSEN (LOGISCH
                         ;  NULL)?
        BCC     DONE     ;JA, DONE
```

Die Befehle ASL und ROL können mit den Bits 6 oder 7 verwendet werden. Ändert sich der Inhalt des VIA-Datenregisters wirklich? Erklären Sie Ihre Antwort.

Aufgabe 2: Zählen der Anzahl der Tasten-Bestätigung

Zweck: Zähle die Anzahl der Tastenbestätigung durch Inkrementieren des Speicherplatzes 0040 nach jedem Schließen.

Beispiel:

Das Drücken der Taste zehnmal nach dem Start des Programmes sollte ergeben:

(0040) = 0A

TASTEN-PRELLEN

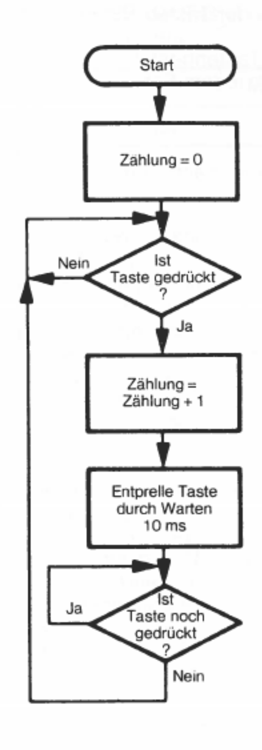
Anmerkung: Um zu zählen, wie oft die Taste betätigt wurde, muß man sich vergewissern, daß jedes Schließen des Tasten-Kontaktes auch tatsächlich einen einzelnen Spannungssprung erzeugt. Nicht jedes Schließen eines mechanischen Tasten-Kontaktes erzeugt auch tatsächlich einen einzelnen Spannungssprung, da die mechanischen Kontakte prellen, bevor sie ihre endgültige Lage erreichen. Man kann einen monostabilen Multivibrator verwenden, um das Prellen zu eliminieren oder man kann dies mittels der Software erledigen.

ENTPRELLEN DURCH SOFTWARE

Das Programm kann die Taste entprellen, indem es nach dem Schließen der Taste wartet. Die erforderliche Verzögerung wird Entprellzeit genannt und ist Teil der Spezifikationen der Taste. Sie ist typisch einige wenige Millisekunden lang. Das Programm sollte die Taste während dieser Periode nicht prüfen, da Fehler infolge des Prellens der Taste entstehen könnten. Das Programm kann entweder in eine Verzögerungs-Routine eintreten, ähnlich der vorher beschriebenen, oder es kann einfach andere Aufgaben während dieses bestimmten Zeitintervalls ausführen.

Auch nach der Entprellung muß das Programm noch warten, bis der momentan geschlossene Kontakt wieder geöffnet wird, bevor es nach weiteren Tasten-Bestätigungen Ausschau hält. Dieses Verfahren vermeidet eine doppelte Zählung. Das folgende Programm verwendet eine Software-Verzögerung von 10 ms, um die Tasten zu entprellen. Man kann auf Wunsch versuchen, die Verzögerung zu variieren oder sie ganz zu eliminieren um festzustellen, was geschieht. Um dieses Programm ablaufen zu lassen, muß man das Verzögerungs-Unterprogramm in den Speicher einbringen, beginnend beim Speicherplatz 0030.

Flußdiagramm:



Quellprogramm:

```
        LDA   #0
        STA   VIAPCR     ;MACHE ALLE STEUER-LEITUNGEN ZU EIN-
                         ;  GÄNGEN
        STA   VIADDRA    ;MACHE PORT-A-LEITUNGEN ZU EIN-
                         ;  GÄNGEN
        STA   $40        ;ZÄHLUNG = 0 ZU ANFANG
CHKCL   LDA   VIAORA
        AND   #MASK      ;IST DIE TASTE GEDRÜCKT?
        BNE   CHKCL      ;NEIN, WARTE BIS DIES DER FALL IST
        INC   $40        ;JA, ADDIERE 1 ZUR ZÄHLUNG
        LDY   #10        ;WARTE 10 MS FÜR ENTPRELLEN
        JSR   DELAY
CHKOP   LDA   VIAORA
        AND   #MASK      ;IST DIE TASTE NOCH IMMER GEDRÜCKT?
        BEQ   CHKOP      ;JA, WARTE AUF LOSLASSEN
        BNE   CHKCL      ;NEIN, HALTE NACH NÄCHSTER
                         ;  BETÄTIGUNG AUSSCHAU
```


Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
0000	A9		LDA	#0
0001	00			
0002	8D		STA	VIAPCR
0003 } 0004 }	VIAPCR			
0005	8D		STA	VIADDRA
0006 } 0007 }	VIADDRA			
0008	85		STA	$40
0009	40			
000A	AD	CHKCL	LDA	VIAORA
000B } 000C }	VIAORA			
000D	29		AND	#MASK
000E	MASK			
000F	D0		BNE	CHKCL
0010	F9			
0011	E6		INC	$40
0012	40			
0013	A0		LDY	#10
0014	0A			
0015	20		JSR	DELAY
0016	30			
0017	00			
0018	AD	CHKOP	LDA	VIAORA
0019 } 001A }	VIAORA			
001B	29		AND	#MASK
001C	MASK			
001D	F0		BEQ	CHKOP
001E	F9			
001F	D0		BNE	CHKCL
0020	E9			

Die drei Befehle, die mit der Markierung CHKOP beginnen, werden zum Warten verwendet, bis die Taste wieder losgelassen wird.

Natürlich benötigen wir nicht wirklich einen VIA für dieses einfache Interface. Ein adressierbares Tristate-Puffer würde die Aufgabe mit wesentlich geringeren Kosten erfüllen.

Ein Kippschalter

Zweck: Anpassung eines einpoligen Kippschalters mit zwei Stellungen an einen Mikroprozessor 6502. Der Kippschalter ist ein mechanischer Baustein, oder in einer von zwei möglichen Stellungen (NC = geschlossen, NO = offen) liegt.

ENTPRELLUNG MIT KREUZ-GEKOPPELTEN NAND-GATTERN

Bild 11-15 zeigt die Schaltung, die zur Anpassung des Schalters erforderlich ist. Wie bei der Taste verwendet der Schalter ein Bit des VIA 6522, das als adressierbarer Puffer dient. Anders als bei der Taste kann der Schalter in einer der beiden Positionen belassen werden. Typische Programm-Aufgaben sind das Bestimmen der Schalter-Position und die Feststellung, ob sich die Position geändert hat. Es kann entweder ein monostabiler Multivibrator mit einer Impulslänge von einigen Millisekunden oder ein paar kreuzgekoppelter NAND-Gatter (siehe Bild 11-16) verwendet werden, um einen mechanischen Schalter zu entprellen.

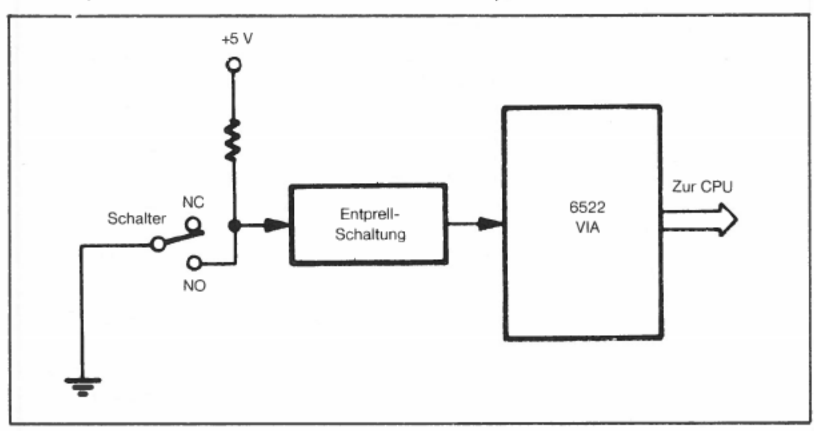


Bild 11-15. Ein Interface für einen Kippschalter.

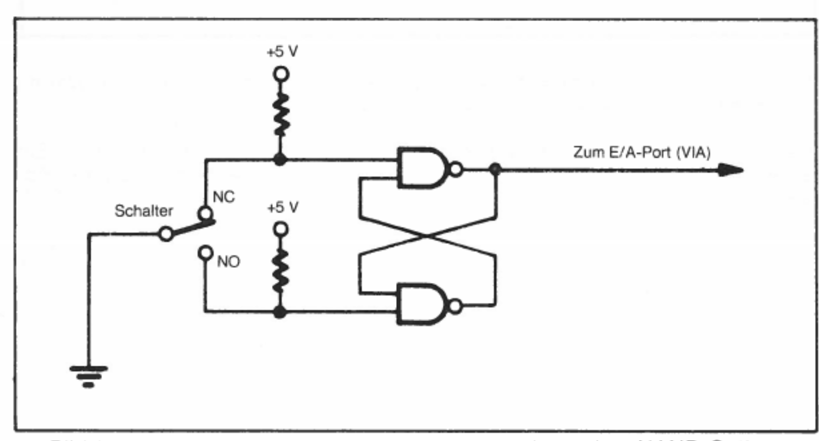


Bild 11-16. Eine Entprell-Schaltung mit kreuzgekoppelten NAND-Gattern.

Die Schaltungen werden einen einzelnen Spannungssprung oder Impuls infolge der Änderung der Schalterpositionen erzeugen, auch wenn der Schalter prellt, bevor er sich in seiner neuen Position befindet.

Programmbeispiel:

Wir wollen zwei Aufgaben unter Verwendung dieser Schaltung ausführen. Diese sind:

1) Setze einen Speicherplatz auf Eins, wenn der Schalter geschlossen ist.

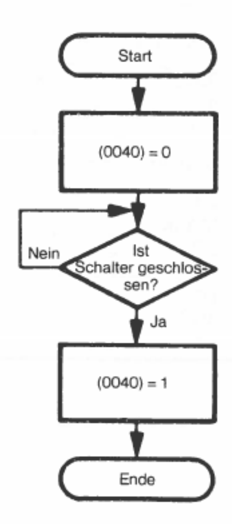
2) Setze einen Speicherplatz auf Eins, wenn sich der Zustand des Schalters ändert.

Aufgabe 1: Warten auf das Schließen des Schalters.

Zweck: Der Speicherplatz 0040 ist Null bis der Schalter geschlossen wird und wird dann auf Eins gesetzt. Das heißt, der Prozessor löscht den Speicherplatz 0040, wartet auf das Schließen des Schalters und setzt dann den Speicherplatz 0040 auf Eins.

Der Schalter könnte als Run/$\overline{\text{Halt}}$ markiert werden, da der Prozessor nicht weiterläuft, bis der Schalter geschlossen wird.

Flußdiagramm:



Quellprogramm:

```
        LDA   #0
        STA   VIAPCR      ;MACHE ALLE STEUER-LEITUNGEN ZU
                          ;  EINGÄNGEN
        STA   VIADDRA     ;MACHE PORT-A-LEITUNGEN
                          ;  ZU EINGÄNGEN
        STA   $40         ;MARKE – NULL
WAITC   LDA   VIAORA      ;LIES SCHALTERSTELLUNG
        AND   #MASK       ;IST DER SCHALTER GESCHLOSSEN (0)?
        BNE   WAITC       ;NEIN, WARTE
        INC   $40         ;JA, MARKE = EINS
        BRK
```

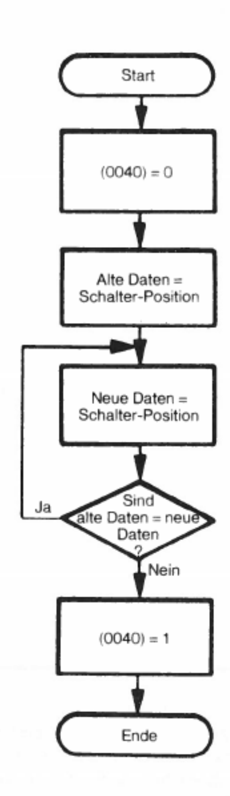
Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
0000	A9		LDA	#0
0001	00			
0002	8D		STA	VIAPCR
0003 } 0004 }	VIAPCR			
0005	8D		STA	VIADDRA
0006 } 0007 }	VIADDRA			
0008	85		STA	$40
0009	40			
000A	AD	WAITC	LDA	VIAORA
000B } 000C }	VIAORA			
000D	29		AND	#MASK
000E	MASK			
000F	D0		BNE	WAITC
0010	F9			
0011	E6		INC	$40
0012	40			
0013	00		BRK	

Aufgabe 2: Warten auf Änderung der Schalterstellung.

Der Speicherplatz 0040 verbleibt Null, bis sich die Schalterstellung ändert und wird dann auf 1 gesetzt. Das heißt, der Prozessor wartet bis zur Änderung der Schalterstellung, und setzt dann den Speicherplatz 0040 auf Eins.

Flußdiagramm:



Quellprogramm:

```
        LDA   #0
        STA   VIAPCR      ;MACHE ALLE STEUERLEITUNGEN ZU
                          ;  EINGÄNGEN
        STA   VIADDRA     ;MACHE PORT A-LEITUNGEN ZU
                          ;  EINGÄNGEN
        STA   $40         ;MARKE = 0
        LDA   VIAORA      ;HOLE ALTE SCHALTERSTELLUNG
        AND   #MASK
        STA   $41
WAITCH  LDA   VIAORA      ;HOLE NEUE SCHALTERSTELLUNG
        AND   #MASK
        CMP   $41         ;SIND DIE NEUE UND DIE ALTE STELLUNG
                          ;  GLEICH?
        BEQ   WAITCH      ;JA, WARTE
        INC   $40         ;NEIN, MARKE = EINS
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A9		LDA	#0
0001	00			
0002	8D		STA	VIAPCR
0003 0004	VIAPCR			
0005	8D		STA	VIADDRA
0006 0007	VIADDRA			
0008	85		STA	$40
0009	40			
000A	AD		LDA	VIAORA
000B 000C	VIAORA			
000D	29		AND	#MASK
000E	MASK			
000F	85		STA	$41
0010	41			
0011	AD	WAITCH	LDA	VIAORA
0012 0013	VIAORA			
0014	29		AND	#MASK
0015	MASK			
0016	C5		CMP	$41
0017	41			
0018	F0		BEQ	WAITCH
0019	F7			
001A	E6		INC	$40
001B	40			
001C	00		BRK	

Eine Subtraktion oder Exklusiv-ODER-Operation könnte den Vergleich (Compare) im Programm ersetzen. Jeder dieser Befehle würde jedoch den Inhalt des Akkumulators ändern. Die Exklusiv-ODER-Operation wäre sehr nützlich, wenn mehrere Schalter mit den gleichen VIA verbunden wären, da es ein Bit für jeden Schalter erzeugen würde, der seinen Zustand geändert hat. Wie würden Sie dieses Programm neu schreiben, damit der Schalter durch die Software entprellt wird?

Ein Schalter mit mehreren Stellungen (Drehschalter, Wahlschalter, oder Daumenrad-Schalter)

Zweck: Anschluß eines Schalters mit mehreren Stellungen an einen Mikroprozessor 6502. Die Leitung, die der Schalterposition entspricht, wird geerdet, während die anderen Leitungen High sind (logisch 1)

Bild 11-17 zeigt die erforderliche Schaltung zum Anschließen eines achtstelligen Schalters. Der Schalter verwendet alle acht Datenbits eines Ports eines VIA. Typische Aufgaben sind das Bestimmen der Schalterposition und Prüfung, ob sich diese Stellung geändert hat oder nicht. Es müssen zwei spezielle Situationen verarbeitet werden.

1) Der Schalter befindet sich zeitweise zwischen den Positionen, so daß keine Leitungen geerdet sind.
2) Der Schalter hat seine endgültige Lage noch nicht erreicht.

Die erste dieser beiden Möglichkeiten kann durch Warten verarbeitet werden, bis alle Eingänge nicht alles Einsen sind, d.h., bis eine Schalterleitung an Masse liegt. Wir können die zweite Möglichkeit durch neuerliches Prüfen der Schalter nach einer Verzögerung (etwa 1 oder 2 Sekunden) verarbeiten und die Eingaben nur akzeptieren, wenn sie gleich bleiben. Diese Verzögerung wird gewöhnlich die Empfindlichkeit des Systems gegenüber den Schaltern nicht beeinflussen. Wir können auch einen anderen Schalter (z.B. einen Lade-Schalter) verwenden, um dem Prozessor mitzuteilen, wann der Wahlschalter gelesen werden sollte.

Programmbeispiele:

Wir wollen zwei Aufgaben für die Schaltung in Bild 11-17 lösen. Diese sind:

a) Beobachten des Schalters, bis er sich in einer endgültigen Position befindet, dann die Position bestimmen und ihren Binärwert in einen Speicherplatz ablegen.
b) Warten auf die Änderung der Schalterposition, dann die neue Position in einen Speicherplatz ablegen.

Wenn sich der Schalter in einer bestimmten Lage befindet, so wird die Leitung von dieser Position über die gemeinsame Leitung an Masse gelegt. Pullup-Widerstände sind gewöhnlich an allen Eingangsleitungen erforderlich, um Probleme zu vermeiden, die durch Rauschen verursacht werden.

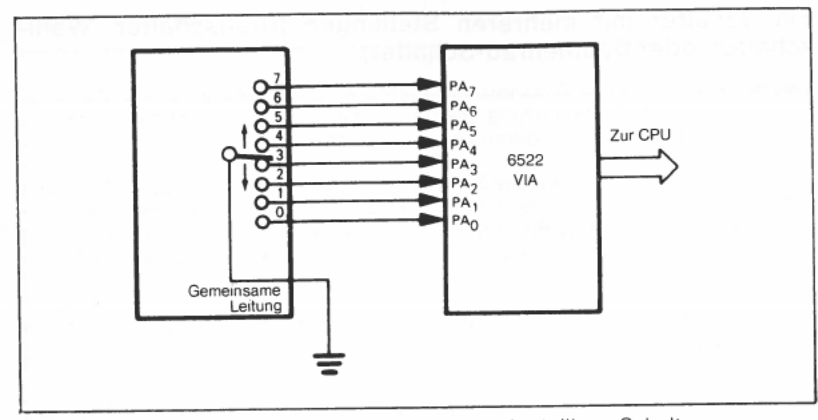


Bild 11-17. Ein Interface für einen mehrstelligen Schalter.

Aufgabe 1: Bestimmen einer Schalterposition

Zweck: Das Programm wartet, bis sich der Schalter in einer endgültigen Position befindet und plaziert dann die Nummer dieser Position in den Speicherplatz 0040.

Tabelle 11-12 enthält den Zusammenhang zwischen den Dateneingängen und den Schalterpositionen.

Tabelle 11-12. Daten-Eingang gegen Schalter-Position.

Switch Position	Data Input	
	Binary	**Hex**
0	11111110	FE
1	11111101	FD
2	11111011	FB
3	11110111	F7
4	11101111	EF
5	11011111	DF
6	10111111	BF
7	01111111	7F

Beachten Sie, daß dieses Schema nicht sehr effizient ist, da es acht Bits benötigt, um zwischen acht verschiedenen Positionen zu unterscheiden.

VERWENDUNG EINES TTL-CODIERERS

Ein TTL- oder MOS-Codierer könnte die Anzahl der benötigten Bits verringern. Bild 11-18 zeigt eine Schaltung, die den TTL-8-zu-3-Codierer 74LS148 verwendet[15]. Wir verbinden die Schalter-Ausgänge in umgekehrter Reihenfolge, da der Baustein 74LS148 Eingänge und Ausgänge besitzt, die aktiv Low sind. Das Ausgangssignal der Codierschaltung ist eine 3-Bit-Darstellung der Schalterposition. Zahlreiche Schalter besitzen derartige Codierer, so daß sie automatisch ein codiertes Ausgangssignal liefern, das gewöhnlich aus einer BCD-Ziffer (in negativer Logik) besteht.

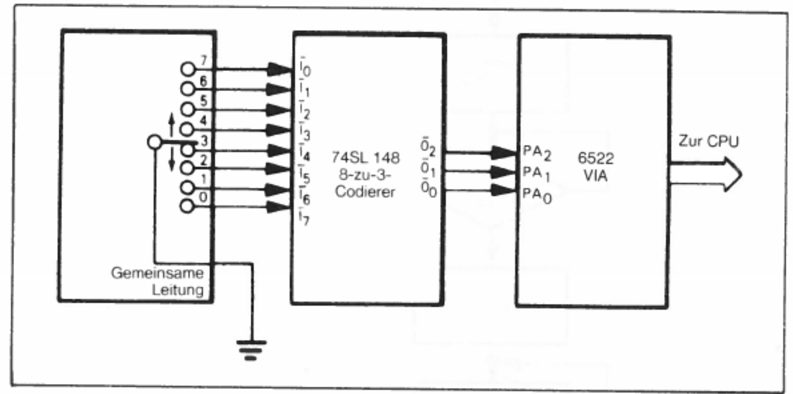
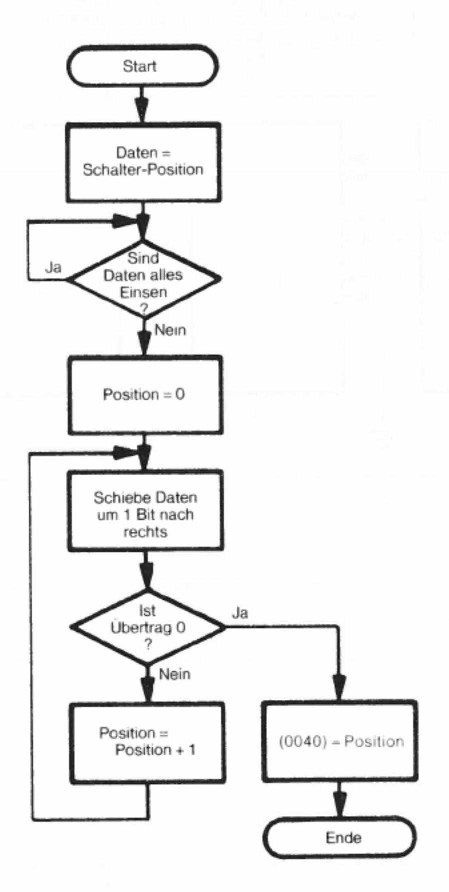


Bild 11-18. Ein mehrstelliger Schalter mit einem Codierer.

Der Codierer erzeugt Ausgangssignale, die aktiv Low sind, so daß beispielsweise die Schalterstellung 5, die mit dem Eingang 2 verbunden ist, ein Ausgangssignal von 2 in negativer Logik (oder 5 in positiver Logik erzeugt. Sie können die doppelte Negation selbst prüfen.

Flußdiagramm:



Quellprogramm:

```
        LDA   #0
        STA   VIAPCR     ;MACHE ALLE STEUER-LEITUNGEN
                         ;  ZU EINGÄNGEN
        STA   VIADDRA    ;MACHE ALLE PORT-A-LEITUNGEN
                         ;  ZU EINGÄNGEN
CHKSW   LDA   VIAORA
        CMP   #$FF       ;IST SCHALTER IN EINER POSITION?
        BEQ   CHKSW      ;NEIN, WARTE AUF EINE SCHALTER-
                         ;  STELLUNG.
        LDX   #0         ;SCHALTER-POSITION = NULL
CHKPOS  ROR   A          ;IST DAS NÄCHSTE BIT EINE GEERDETE
                         ;  POSITION?
        BCC   DONE       ;JA, SCHALTERSTELLUNG GEFUNDEN
        INX              ;NEIN, INKREMENTIERE SCHALTER-
                         ;  STELLUNG
        JMP   CHKPOS
DONE    STX   $40        ;SPEICHERE SCHALTERSTELLUNG
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A9		LDA	#0
0001	00			
0002	8D		STA	VIAPCR
0003 0004	VIAPCR			
0005	8D		STA	VIADDRA
0006 0007	VIADDRA			
0008	AD	CHKSW	LDA	VIAORA
0009 000A	VIAORA			
000B	C9		CMP	#$FF
000C	FF			
000D	F0		BEQ	CHKSW
000E	F9			
000F	A2		LDX	#0
0010	00			
0011	6A	CHKPOS	ROR	A
0012	90		BCC	DONE
0013	04			
0014	E8		INX	
0015	4C		JMP	CHKPOS
0016	11			
0017	00			
0018	86	DONE	STX	$40
0019	40			
001A	00		BRK	

Nehmen Sie an, daß ein fehlerhafter Schalter oder ein defekter VIA immer ein Ausgangssignal liefern würde, das FF_{16} ist. Wie könnten Sie dieses Programm abändern, so daß es diesen Fehler feststellen würde?

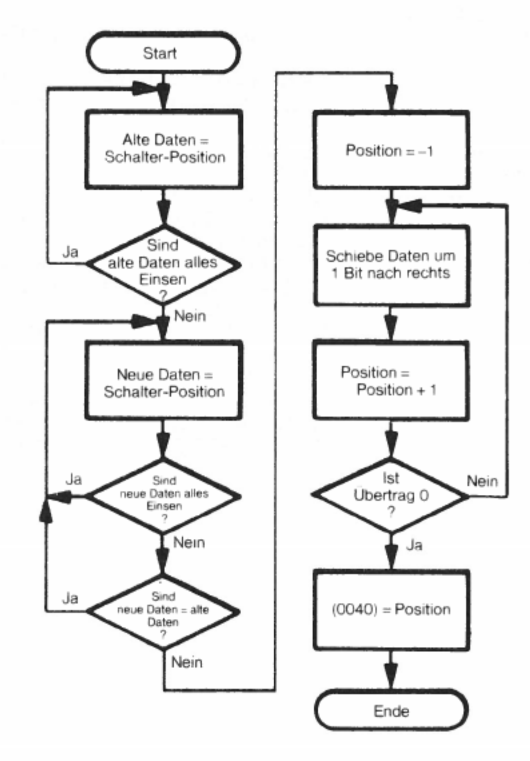
Dieses Programm könnte leicht anders aufgebaut werden, um es kürzer und schneller zu machen – so wie verschiebbar. Eine Option wäre das Ersetzen von JMP CHKPOS durch BCS CHKPOS. Weshalb ist dies möglich und welche Verbesserungen ergeben sich daraus? Eine weitere Option wäre das Ändern der Anfangsbedingungen, so daß nur ein Sprungbefehl erforderlich wäre. Wie würden Sie dies ausführen? Hinweis: Beginnen Sie mit FF_{16} im Indexregister X und inkrementieren Sie X vor dem Verschieben des Akkumulators.

Dieses Beispiel nimmt an, daß der Schalter mittels Hardware entprellt wurde. Wie würden Sie dieses Programm ändern, damit die Entprellung durch die Software erfolgt?

Aufgabe 2: Warten auf eine Änderung der Schalterposition

Zweck: Das Programm wartet auf eine Änderung der Schalterposition und plaziert die neue Position (decodiert) in den Speicherplatz 0040. Das Programm wartet, bis der Schalter seine neue Lage erreicht.

Flußdiagramm:



Quellprogramm:

```
        LDA   #0
        STA   VIAPCR       ;MACHE ALLE STEUER-LEITUNGEN ZU
                           ;  EINGÄNGEN
        STA   VIADDRA      ;MACHE ALLE PORT-A-LEITUNGEN ZU
                           ;  EINGÄNGEN
CHKFST  LDA   VIAORA
        CMP   #$FF         ;BEFINDET SICH DER SCHALTER IN EINER
                           ;  POSITION?
        BEQ   CHKFST       ;NEIN, WARTE AUF DIE POSITION
        TAX                ;BEWAHRE ALTE POSITION AUF
CHKSEC  LDA   VIAORA
        CMP   #$FF         ;BEFINDET SICH DER SCHALTER IN EINER
                           ;  POSITION
        BEQ   CHKSEC       ;NEIN, WARTE AUF POSITION
        CPX   VIAORA       ;IST DIE POSITION DIE GLEICHE WIE
                           ;  VORHER
        BEQ   CHKSEC       ;JA, WARTE AUF EINE ÄNDERUNG
        RORB               ;IST DAS NÄCHSTE BIT EINE GEERDETE
                           ;  POSITION?
        LDX   #$FF         ;NEIN, STARTE SCHALTERSTELLUNG BEI -1
CHKPOS  INX                ;SCHALTER-POSITION = SCHALTER-
                           ;  POSITION+1
        ROR   A            ;LIEGT NÄCHSTES BIT AN MASSE?
        BCS   CHKPOS       ;NEIN, HALTE WEITER AUSSCHAU
        STA   $40          ;SPEICHERE SCHALTERPOSITION
        BRK
```


Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A9		LDA	#0
0001	00			
0002	8D		STA	VIAPCR
0003, 0004	VIAPCR			
0005	8D		STA	VIADDRA
0006, 0007	VIADDRA			
0008	AD	CHKFST	LDA	VIAORA
0009, 000A	VIAORA			
000B	C9		CMP	#$FF
000C	FF			
000D	F0		BEQ	CHKFST
000E	F9			
000F	AA		TAX	
0010	AD	CHKSEC	LDA	VIAORA
0011, 0012	VIAORA			
0013	C9		CMP	#$FF
0014	FF			
0015	F0		BEQ	CHKSEC
0016	F9			
0017	EC		CPX	VIAORA
0018, 0019	VIAORA			
001A	F0		BEQ	CHKSEC
001B	F4			
001C	A2		LDX	#$FF
001D	FF			
001E	E8	CHKPOS	INX	
001F	6A		ROR	A
0020	B0		BCS	CHKPOS
0021	FC			
0022	86		STX	$40
0023	40			
0024	00		BRK	

Eine andere Methode zur Bestimmung, ob sich der Schalter in einer Stellung befindet ist:

```
CHKSW   INC   VIAORA
        BEQ   CHKSW
```

Wie arbeitet diese? Was geschieht mit den Eingangsdaten?

Schreiben Sie das Programm zur Verwendung der alternativen Methode. Wieviel weniger Speicher ist erforderlich?

Eine einzelne LED

Zweck: Anpassung einer licht-emittierenden Diode an einen Mikroprozessor 6502. Die LED kann so angeordnet werden, daß sie entweder durch eine Null oder eine Eins eingeschaltet wird.

LED-STEUERUNG

Bild 11-19 zeigt die Schaltung, die zum Anschluß einer LED erforderlich ist. Die LED leuchtet, wenn ihre Anode positiv gegenüber ihrer Kathode ist (Bild 11-19 a). Daher kann man die LED entweder dadurch zum Leuchten bringen indem man die Kathode an Masse legt, und der Computer eine Eins an die Anode liefert (Bild 11-19 b) oder durch Verbinden der Anode mit +5 Volt, wobei der Computer eine Null an die Kathode liefert (Bild 11-19 c). Die Steuerung der Kathode ist die am häufigsten angewandte Lösung. Die LED ist am hellsten, wenn sie mit einem gepulsten Strom von etwa 10 bis 50 mA gespeist wird, der einige hundertmal pro Sekunde zugeführt wird. LEDs besitzen eine sehr kurze Einschaltzeit (im Mikrosekunden-Bereich), so daß sie sehr gut für Multiplexen(der Betrieb mehrerer LEDs von einem einzelnen Port) geeignet sind. LED-Schaltungen benötigen gewöhnlich periphere Bausteine oder Transistor-Treiber und strom-begrenzende Widerstände. MOS-Schaltungen können normalerweise LEDs nicht direkt steuern und sie hell genug zum Leuchten bringen.

Achtung: Der VIA besitzt einen Ausgangs-Zwischenspeicher an jedem Port. Der Port B wird jedoch normalerweise für Ausgaben verwendet, da er etwas höhere Treibermöglichkeit besitzt. Insbesondere sind die Ausgänge des Ports B in der Lage, Darlington-Transistoren anzusteuern (er liefert mindestens 3.0 mA bei 1.5 V). Darlington-Transistoren sind Transistoren mit sehr hoher Verstärkung, die in der Lage sind, große Ströme mit hoher Geschwindigkeit zu schalten. Sie sind sehr nützlich bei der Ansteuerung von Spulen, Relais und anderen Bausteinen.

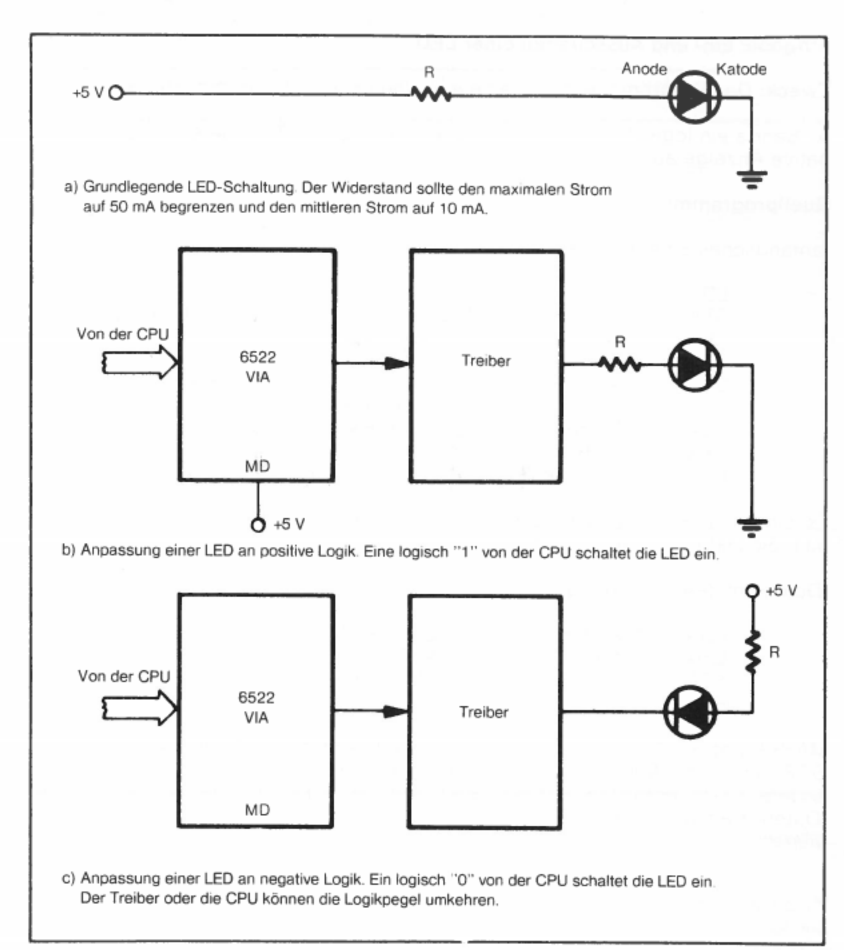


Bild 11-19. Anpassung einer LED.

Aufgabe: Ein- und Ausschalten einer LED

Zweck: Das Programm schaltet eine einzelne LED entweder ein oder aus.

A. Sende ein logisch 1 zur LED (schalte eine positive Anzeige ein oder eine negative Anzeige aus).

Quellprogramm:

(anfängliches Einstellen der Daten)

```
LDA   #0
STA   VIAPCR     ;MACHE ALLE STEUER-LEITUNGEN ZU
                 ;  EINGÄNGEN
LDA   #$FF
STA   VIADDRB    ;MACHE PORT-B-LEITUNGEN ZU
                 ;  AUSGÄNGEN
LDA   #MASKP     ;HOLE DATEN FÜR LED
STA   VIAORB     ;SENDE SIE ZUR LED
BRK
```

Es wird die B Seite der VIA verwendet, da diese gepuffert ist. Die CPU kann deshalb die Daten vom Ausgangsport lesen.

(Daten auf den neuesten Stand bringen)

```
LDA   VIAORB     ;HOLE DIE ALTEN DATEN
ORA   #MASKP     ;SCHALTE LED-BIT EIN
STA   VIAORB     ;SENDE DATEN ZUR LED
BRK
```

MASKP hat ein 1-Bit in der LED-Position und andernfalls Nullen. Eine logische ODERierung mit MASKP beeinflußt die anderen Bit-Positionen nicht, die Bits für andere LEDs enthalten können. Beachten Sie, daß wir die VIA-Ausgangs-(Daten)-Register lesen können, auch wenn die Anschlüsse als Ausgänge dienen.

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)	
(Erzeugen der Anfangsdaten)			
0000	A9	LDA	#0
0001	00		
0002	8D	STA	VIAPCR
0003 } 0004 }	VIAPCR		
0005	A9	LDA	#$FF
0006	FF		
0007	8D	STA	VIADDRB
0008 } 0009 }	VIADDRB		
000A	A9	LDA	#MASKP
000B	MASKP		
000C	8D	STA	VIAORB
000D } 000E }	VIAORB		
000F	00	BRK	
(Daten auf den neuesten Stand bringen)			
0010	AD	LDA	VIAORB
0011 } 0012 }	VIAORB		
0013	09	ORA	#MASKP
0014	MASKP		
0015	8D	STA	VIAORB
0016 } 0017 }	VIAORB		
0018	00	BRK	

B. Sende eine logische Null zur LED (Abschalten einer positiven Anzeige oder Einschalten einer negativen Anzeige).

Die Unterschiede sind, daß MASKP durch sein logisches Komplement MASKN und ORA#MASKP durch AND#MASKN ersetzt werden muß. MASKN hat ein Null-Bit in der LED-Position und sonst Einsen. Logische UNDierung mit MASKN beeinflußt die anderen Bit-Positionen nicht.

7-Segment-LED-Anzeige

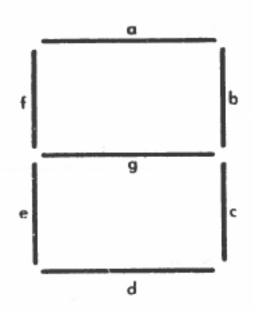
Zweck: Anpassung einer 7-Segment-LED-Anzeige an einen Mikroprozessor 6502. Die Anzeige kann entweder eine gemeinsame Anode (negative Logik) oder eine gemeinsame Kathode (positive Logik) besitzen.

Schaltung:

ANZEIGEN MIT GEMEINSAMER ANODE ODER MIT GEMEINSAMER KATHODE

Bild 11-20 zeigt die Schaltung, die zur Anpassung einer 7-Segment-LED-Anzeige erforderlich ist. Jedes Segment kann ein, zwei oder mehr LEDs besitzen, die in der gleichen Weise angeordnet sind. Es gibt zwei Möglichkeiten für die Verbindung von LEDs. Bei einer Anordnung werden alle Kathoden gemeinsam an Masse geführt (siehe Bild 11-21 a). Dies ist eine Anzeige mit "gemeinsamer Kathode", und eine logische Eins an einer Anode bringt dieses Segment zum Leuchten. Bei einer anderen Möglichkeit werden alle Anoden gemeinsam an eine positive Betriebsspannung geführt (siehe Bild 11-21 b). Dies ist eine Anzeige mit "gemeinsamer Anode", und eine logische Null an einer Kathode läßt dieses Segment leuchten. Daher verwendet die Anzeige mit gemeinsamer Kathode positive Logik und die Anzeige mit gemeinsamer Anode negative Logik. Jede Anzeige benötigt entsprechende Treiber und Widerstände.

Die gemeinsame Leitung von der Anzeige wird entweder an Masse oder an +5 Volt gelegt. Die Anzeige-Segmente werden gewöhnlich folgendermaßen bezeichnet:



Anmerkung: Die 7-Segment-Anzeige ist weit verbreitet, da sie die kleinste Anzahl von getrennt gesteuerten Segmenten enthält, die eine gut erkennbare Darstellung aller Dezimalzahlen liefert (siehe Bild 11-22 und Tabelle 11-13). Mit 7-Segment-Anzeigen kann man auch einige Buschstaben und andere Zeichen erzeugen (siehe Tabelle 11-14). Bessere Darstellungen benötigen eine wesentlich größere Anzahl von Segmenten und aufwendigere Schaltungen[16]. Da 7-Segment-Anzeigen so populär sind, sind preisgünstige Decoder/Treiber für 7-Segment-Anzeigen in großem Umfang verfügbar. Der populärste Baustein ist der Treiber für gemeinsame Anoden 7447 und der Treiber für gemeinsame Kathoden 7448[17]. Diese Bausteine besitzen Eingänge für Lampen-Tests (mit denen alle Segmente eingeschaltet werden) und Austast-Eingänge und Ausgänge (für das Ausblenden von führenden oder nachstehenden Nullen).

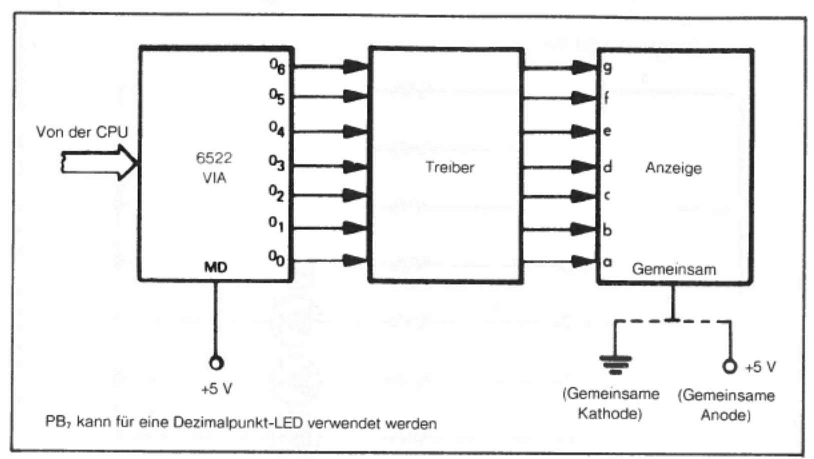


Bild 11-20. Anschluß einer Sieben-Segment-Anzeige.

Tabelle 11-13. Sieben-Segment-Darstellung von Dezimalzahlen.

Zahl	Hexadezimale Darstellung	
	Gemeinsame Kathode	Gemeinsame Anode
0	3F	40
1	06	79
2	5B	24
3	4F	30
4	66	19
5	6D	12
6	7D	02
7	07	78
8	7F	00
9	67	18

Bit 7 ist immer null und die anderen sind g, f, e, d, c, b und a in absteigender Reihenfolge der Signifikanz.

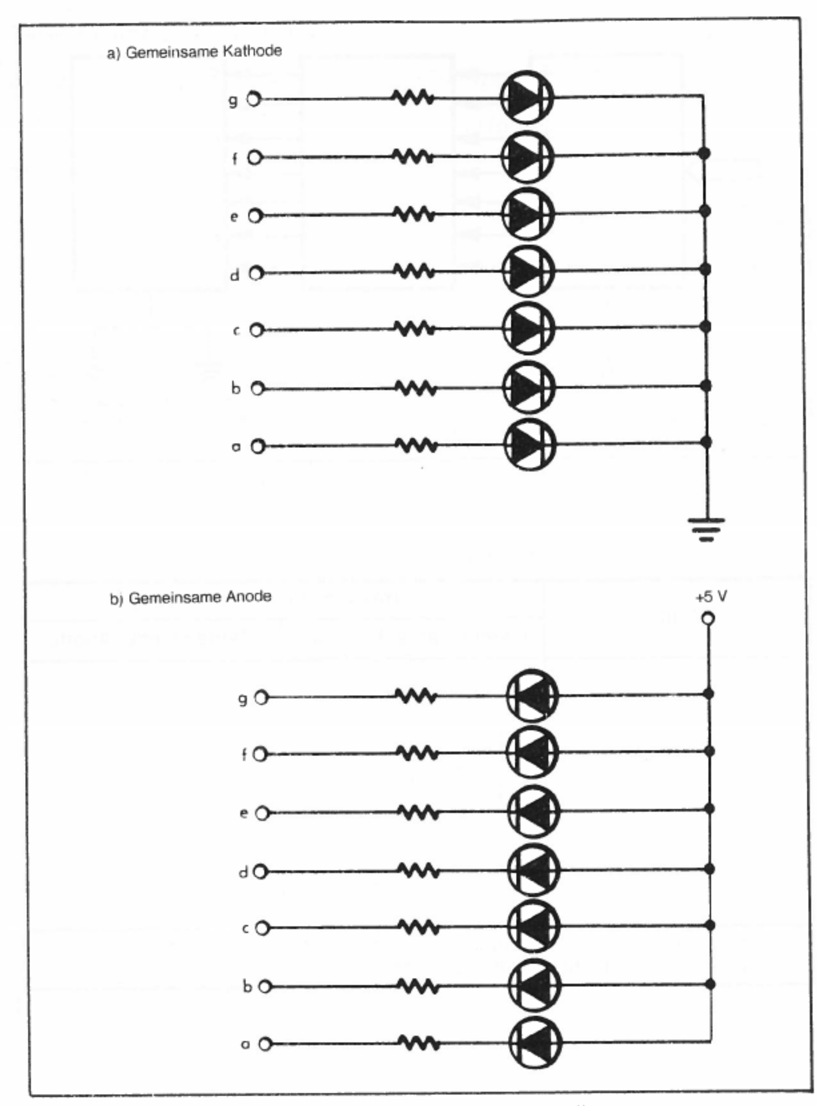


Bild 11-21. Sieben-Segment-Darstellung

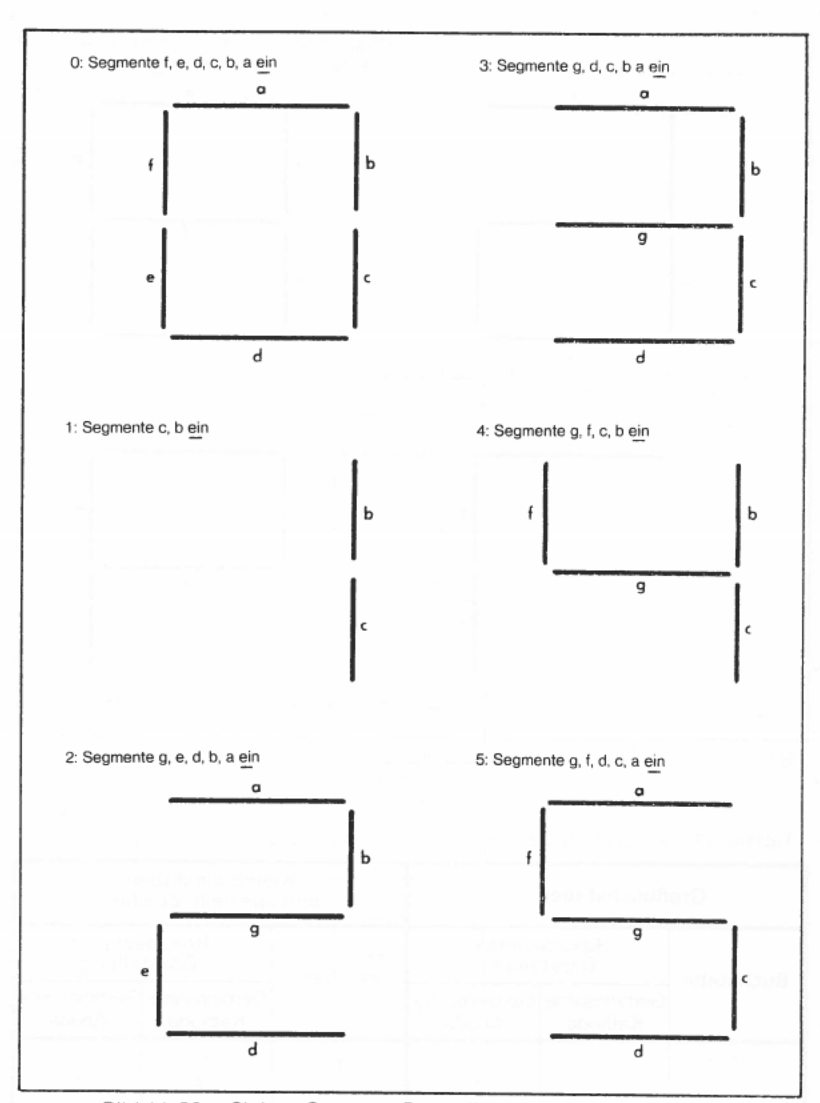


Bild 11-22. Sieben-Segment-Darstellung von Dezimalziffern.

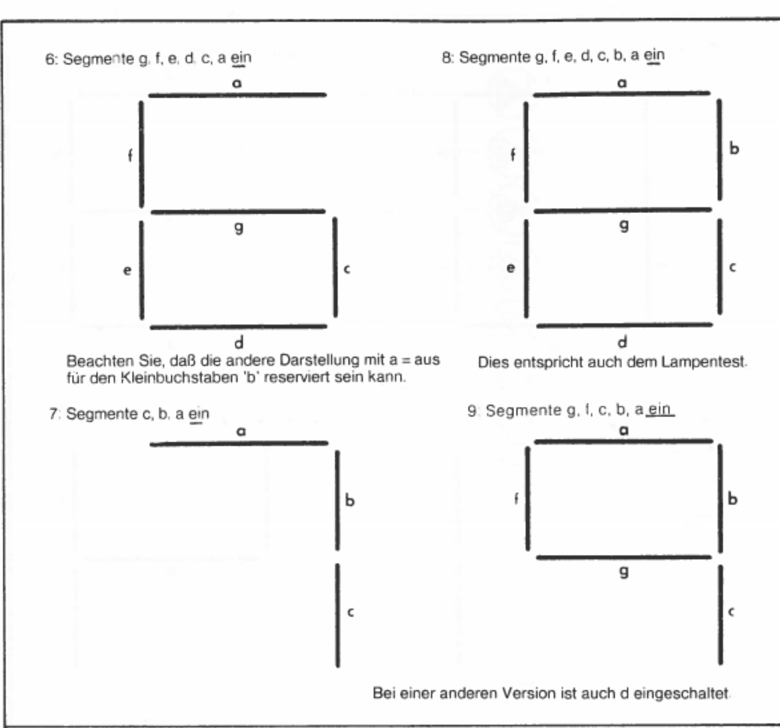


Bild 11-22. Sieben-Segment-Darstellung von Dezimalziffern (Fortsetzung).

Tabelle 11-14. Sieben-Segment-Darstellung von Buchstaben und Symbolen.

Großbuchstaben			Kleinbuchstaben und spezielle Zeichen		
Buchstabe	Hexadezimale Darstellung		Zeichen	Hexadezimale Darstellung	
	Gemeinsame Kathode	Gemeinsame Anode		Gemeinsame Kathode	Gemeinsame Anode
A	77	08	b	7C	03
C	39	46	c	58	27
E	79	06	d	5E	21
F	71	0E	h	74	0B
H	76	09	n	54	2B
I	06	79	o	5C	23
J	1E	61	r	50	2F
L	38	47	u	1C	63
O	3F	40	-	40	3F
P	73	0C	?	53	2C
U	3E	41			
Y	66	19			

Aufgabe 1: Anzeige einer Dezimalziffer

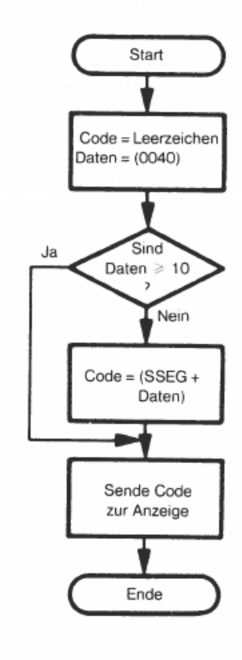
Zweck: Zeige an Inhalt des Speicherplatzes 0040 auf einer 7-Segment-Anzeige an, wenn er eine Dezimalziffer enthält. Taste andernfalls Anzeige aus.

Beispiele:

a. (0040) = 05
Resultat ist 5 auf der Anzeige

b. (0040) = 66
Das Resultat ist eine dunkel getastete Anzeige

Flußdiagramm:



Quellprogramm:

```
         LDA    #0
         LDA    VIAPCR      ;MACHE ALLE STEUER-LEITUNGEN ZU
                            ;  EINGÄNGEN
         LDA    #$FF
         STAA   VIADDRB     ;MACHE PORT-B-LEITUNGEN ZU
                            ;  AUSGÄNGEN
         LDA    #BLANK      ;HOLE AUSTAST(BLANK)-CODE
         LDX    $40         ;HOLE DATEN
         CPX    #10         ;SIND DATEN 10 ODER GRÖSSER?
         BCS    DSPLY       ;JA, TASTE ANZEIGE DUNKEL
         LDA    SSEG,X      ;WANDLE DATEN IN 7-SEGMENT-CODE UM
DSPLY    STA    VIAORB      ;SENDE CODE ZUR ANZEIGE
         BRK
```

BLANK ist 00 für eine Anzeige mit gemeinsamer Kathode, FF für eine Anzeige mit gemeinsamer Anode. Ein weiteres Verfahren bestünde darin, den BLANK-Code an das Ende der Tabelle zu legen und alle ungeeigneten Daten durch 10 zu ersetzen, d.h. die Befehle nach STA VIADDRB sind:

```
         LDX    #40         ;HOLE DATEN
         CMX    #10         ;SIND DATEN 10 ODER GRÖSSER?
         BCC    CNVRT
         LDX    #10         ;JA, ERSETZE SIE DURCH 10
CNVRT    LDA    SSEG,X      ;WANDLE DATEN IN 7-SEGMENT-CODE UM
```

Die Tabelle SSEG ist entweder eine Darstellung der Dezimalziffern für die gemeinsame Kathode oder gemeinsame Anode aus Tabelle 11-13.

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
0000	A9		LDA	#0
0001	00			
0002	8D		STA	VIAPCR
0003 } 0004 }	VIAPCR			
0005	A9		LDA	#$FF
0006	FF			
0007	8D		STA	VIADDRB
0008 } 0009 }	VIADDRB			
000A	A9		LDA	#BLANK
000B	BLANK			
000C	A6		LDX	$40
000D	40			
000E	E0		CPX	#10
000F	0A			
0010	B0		BCS	DSPLY
0011	02			
0012	B5		LDA	SSEG,X
0013	20			
0014	8D	DSPLY	STA	VIAORB
0015 } 0016 }	VIAORB			
0017	00		BRK	
0020-0029		SSEG	(Sieben-Segment-Code-Tabelle)	

Mehrere Anzeigen können gemultiplext werden, wie in Bild 11-23 gezeigt ist. Ein kurzer Tastimpuls auf der Stuerleitung CB2 taktet den Zähler und leitet die Daten zur nächsten Anzeige. RESET startet den Dezimal-Zähler bei 9, so daß die erste Ausgabe-Operation den Zähler löscht und die Daten zur ersten Anzeige führt.

Das folgende Programm verwendet die Verzögerungs-Routine zum Pulsen jeder der zehn Anzeigen mit gemeinsamer Kathode während der Dauer von 1 ms.

Aufgabe 2: Anzeige von zehn Dezimalziffern

Zweck: Zeige den Inhalt der Speicherplätze 0040 bis 0049 auf zehn 7-Segment-Anzeigen an, die durch einen Zähler und einen Decoder gemultiplext werden. Die höchstwertige Stelle liegt in 0049.

Beispiel:

(0040) = 66
(0041) = 3F
(0042) = 7F
(0043) = 7F
(0044) = 06
(0045) = 5B
(0046) = 07
(0047) = 4F
(0048) = 6D
(0049) = 7D
Die Anzeige liest 6537218804

Die Schaltung in Bild 11-23 verwendet das VIA-Quittierungssignal CB2 als einen kurzen Ausgangsimpuls, um das Auftreten eines Datentransfers anzuzeigen.

Quellprogramm:

```
        LDA   #$FF
        STA   VIADDRB      ;MACHE PORT-B-LEITUNGEN ZU
                           ;  AUSGÄNGEN
        LDA   #%10100000
        STA   VIAPCR       ;LIEFERE "DATA-READY"-IMPULS
SCAN    LDX   #10          ;ANZAHL DER ANZEIGEN = 10
DSPLY   LDA   $3F,X        ;HOLE DATEN FÜR ANZEIGE
        STA   VIAORB       ;SENDE DATEN ZUR ANZEIGE
        JSR   DELAY        ;WARTE 1 MS
        DEX
        BNE   DSPLY        ;ZÄHLE ANZEIGEN
        BRA   SCAN         ;STARTE WEITERE ABTASTUNG
```

Das periphere Steuerregister-Bit 7 = 1, um CB2 zu einem Ausgang zu machen, Bit 6 = 1, um einen Impuls zu erzeugen, und Bit 3 = 1, um einen kurzen Tastimpuls zu erzeugen. Wir haben angenommen, daß das Unterprogramm DELAY so modifiziert wurde, daß es ein transparentes Warten von 1 ms liefert.

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A9		LDA	#$FF
0001	FF			
0002	8D		STA	VIADDRB
0003, 0004	VIADDRB			
0005	A9		LDA	#%10100000
0006	A0			
0007	8D		STA	VIAPCR
0008, 0009	VIAPCR			
000A	A2	SCAN	LDX	#10
000B	0A			
000C	B5	DSPLY	LDA	$3F,X
000D	3F			
000E	8D		STA	VIAORB
000F, 0010	VIAORB			
0011	20		JSR	DELAY
0012	30			
0013	00			
0014	CA		DEX	
0015	D0		BNE	DSPLY
0016	F5			
0017	F0		BEQ	SCAN
0018	F1			

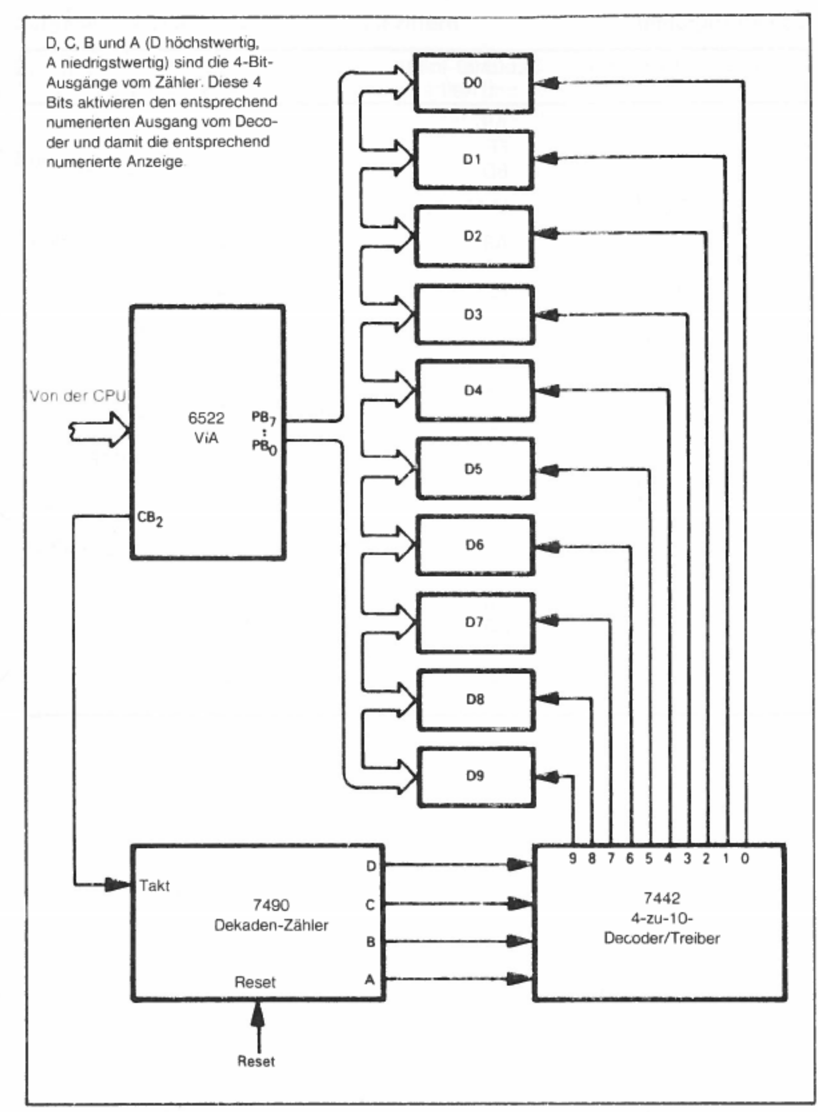


Bild 11-23. Gemultiplexte Sieben-Segment-Anzeigen.

AUFGABEN

1) Eine Ein-Aus-Taste

Zweck: Jedes Schließen der Taste komplementiert (invertiert) alle Bits im Speicherplatz 0040. Der Speicherplatz enthält anfangs null. Das Programm sollte die Taste kontinuierlich prüfen und den Speicherplatz 0040 bei jedem Schließen komplementieren. Man kann auch stattdessen einen Anzeige-Ausgangsport komplementieren, damit die Ergebnisse leichter zu sehen sind.

Beispiel:

Der Speicherplatz 0040 enthält anfangs null.

Das erste Schließen der Taste ändert den Inhalt des Speicherplatzes 0040 in FF_{16}, etc. Es werde angenommen, daß die Taste durch Hardware entprellt wird. Wie könnten Sie Entprellen in Ihr Programm aufnehmen?

2) Entprellen eines Schalters durch Software

Zweck: Entprellen eines mechanischen Schalters durch Warten, bis zwei Lesungen, die eine Entprellzeit voneinander entfernt sind, das gleiche Ergebnis liefern. Es werde angenommen, daß sich die Entprellzeit (in ms) im Speicherplatz 0040 befindet und die Schalterstellung in den Speicherplatz 0041 plaziert wird.

Beispiel:

(0040) = 03 bewirkt, daß das Programm 3 ms zwischen den Lesungen wartet.

3) Steuerung für einen Drehschalter

Zweck: Ein weiterer Schalter dient als Ladeschalter für einen nicht codierten Drehschalter mit vier Stellungen. Die CPU wartet, bis der Ladeschalter geschlossen (null) ist, und liest dann die Stellung des Drehschalters. Dieses Verfahren gestattet dem Operator das Drehen des Drehschalters in seine Endposition, bevor die CPU versucht ihn zu lesen. Das Programm sollte die Stellung des Drehschalters in den Speicherplatz 0040 plazieren. Der Ladeschalter ist mit Software zu entprellen.

Beispiel:

Bringe Drehschalter in Position 2. Schließe Ladeschalter.

Ergebnis: (0042) = 02

4) Darstellung der Schalterpositionen auf Lampen

Zweck: Bei einem Satz von acht Schaltern sollte deren Position durch acht LEDs dargestellt werden. Das heißt, wenn die Schalter geschlossen (null) sind, sollte die LED eingeschaltet sein. Andernfalls sollte die LED ausgeschaltet sein. Es werde angenommen, daß der Ausgangs-Port der CPU mit den Kathoden der LEDs verbunden ist.

Beispiel:

SCHALTER 0 GESCHLOSSEN
SCHALTER 1 OFFEN
SCHALTER 2 GESCHLOSSEN
SCHALTER 3 OFFEN
SCHALTER 4 OFFEN
SCHALTER 5 GESCHLOSSEN
SCHALTER 6 GESCHLOSSEN
SCHALTER 7 OFFEN

Ergebnis:

LED 0 EIN
LES 1 AUS
LED 2 EIN
LED 3 AUS
LED 4 AUS
LED 5 EIN
LED 6 EIN
LED 7 AUS

Wie würden Sie das Programm ändern, damit ein Schalter, der mit Bit 7 der Seite A des VIA Nr. 2 verbunden ist, bestimmt, ob die Anzeigen aktiv sind oder nicht (d.h., wenn der Steuerschalter geschlossen ist, geben die mit Seite B verbundenen Anzeigen den Zustand der mit Seite A verbundenen Schalter wieder. Wenn der Steuerschalter offen ist, sind die Anzeigen immer abgeschaltet)? Ein Steuerschalter ist nützlich, wenn die Anzeigen den Bedienenden stören können, wie etwa in einem Flugzeug.

Wie würden Sie das Programm ändern, damit der Steuerschalter eine Ein-Aus-Taste ist? Das heißt, jedes Schließen kehrt den vorhergehenden Zustand der Anzeigen um. Es werde angenommen, daß die Anzeigen im aktiven Zustand beginnen und das das Programm die Taste prüft und entprellt, bevor es Daten zu den Anzeigen schickt.

5) Zählung auf einer 7-Segment-Anzeige

Zweck: Das Programm sollte von 0 bis 9 kontinuierlich auf einer 7-Segment-Anzeige zählen, beginnend mit null.

Hinweis: Versuchen Sie unterschiedliche Längen für die zeitliche Steuerung der Anzeigen und beachten Sie, was geschieht. Wann wird die Anzeige sichtbar? Was geschieht, wenn die Anzeige während eines Teiles der Zeit ausgetastet wird?

KOMPLEXERE E/A-BAUSTEINE

Komplexere E/A-Bausteine unterscheiden sich von einfachen Tastaturen, Schaltern und Anzeigen in folgendem:

1) Sie transferieren Daten mit höheren Geschwindigkeiten.
2) Sie können ihren eigenen internen Takt und eigene zeitliche Steuerung besitzen.
3) Sie erzeugen eine Status-Information und benötigen Steuer-Informationen, und transferieren Daten.

Infolge ihrer höheren Datenraten muß man diese E/A-Bausteine entsprechend handhaben. Wenn der Prozessor nicht den geeigneten Service liefert, kann das System Eingangsdaten übersehen oder falsche Ausgangsdaten erzeugen. Man arbeitet daher unter wesentlich exakteren Einschränkungen, als dies bei einfachen Bausteinen der Fall ist. Unterbrechungen sind ein bequemes Verfahren für die Handhabung von komplexen E/A-Bausteinen, die wir in Kapitel 12 sehen werden.

SYNCHRONISATION MIT E/A-BAUSTEINEN

Periphere Geräte wie Tastaturen, Fernschreiber, Kassetten und Floppy-Disks erzeugen ihre eigene interne zeitliche Steuerung. Diese Bausteine liefern Ströme von Daten, getrennt durch spezielle zeitliche Intervalle. Der Computer muß die anfängliche Eingabe- oder Ausgabe-Operation mit dem peripheren Takt synchronisieren und dann die entsprechenden Intervalle zwischen aufeinanderfolgenden Operationen liefern. Eine einfache Verzögerungs-Schleife, wie die früher gezeigte, kann die Zeit-Intervalle erzeugen. Die Synchronisation kann eine oder mehrere der folgenden Verfahren benötigen:

1) Das Ausschau-Halten nach einem Übergang auf einer Takt- oder Tastleitung, die vom peripheren Baustein für die zeitliche Steuerung zur Verfügung gestellt wird. Das einfachste Verfahren besteht darin, den Tast-Impuls an eine VIA-Steuerleitung zu führen und auf eine Änderung in dem entsprechenden Bit des VIA-Steuerregisters zu warten.

2) Auffinden der Mitte des Zeit-Intervalles, währenddem die Daten stabil sind. Man sollte es vorziehen, den Wert der Daten in der Mitte des Impulses zu bestimmen, anstatt an den Flanken, an denen sich die Daten ändern können. Das Auffinden der Mitte erfordert eine Verzögerung um die Hälfte eines Übertragungs-Intervalles (Bit-Zeit) nach der Flanke. Das Abtasten (Sampeln) der Daten in der Mitte bedeutet auch, daß kleine zeitliche Fehler nur wenig Einfluß auf die Genauigkeit des Empfanges haben.

3) Erkennen eines speziellen Startcodes. Dies ist leicht, wenn der Code aus einem einzelnen Bit besteht, oder wenn wir einige zeitliche Informationen besitzen. Das Verfahren ist komplexer, wenn der Code lang ist und zu jeder Zeit beginnen könnte. Es wird ein Verschieben erforderlich sein um zu bestimmen, wo der Sender seine Bits, Zeichen oder Nachrichten startet (dies wird auch häufig ein Suchen nach der richtigen "Rahmung"(Framing) genannt.

4) Wiederholtes Abtasten der Daten. Dies verringert die Wahrscheinlichkeit des Empfanges falscher Daten auf verrauschten Leitungen. Eine Majoritäts-Logik (wie etwa "best 3 out of 5" oder "5 out of 8") kann zur Feststellung des tatsächlichen Datenwertes verwendet werden.

Der Empfang ist natürlich wesentlich schwieriger als das Senden, da die peripheren Geräte den Empfang steuern und der Computer die zeitlichen Informationen interpretieren muß, die durch den peripheren Baustein erzeugt werden. Bei der Sendung liefert der Computer die entsprechende zeitliche Steuerung und Formatierung für einen speziellen peripheren Baustein.

STEUER- UND STATUS-INFORMATIONEN

Periphere Bausteine können andere Informationen neben den Daten und der zeitlichen Steuerung benötigen oder liefern. Wir bezeichnen die anderen Informationen, die vom Computer gesendet werden, als "Steuer-Informationen". Sie können die Art der Operation, Start oder Stop des Vorganges, Taktregister, Freigabe-Puffer wählen, ebenso Formate oder das Übertragungs-Protokoll, Anzeigen für den Operator liefern, Operationen zählen oder die Art und die Priorität einer Operation identifizieren. Wir nennen andere Informationen, die von den peripheren Bausteinen gesendet werden, "Status-Informationen". Sie können die Art der Operation, die Bereitschaft der Bausteine, das Vorhandensein von Fehlerbedingungen, das Format des verwendeten Protokolls und andere Zustände oder Bedingungen anzeigen.

Der Computer verarbeitet Steuer- und Status-Informationen ebenso wie Daten. Diese Informationen ändern sich selten, obwohl in Wirklichkeit Daten mit hoher Geschwindigkeit übertragen werden können. Die Steuer- oder Status-Informationen können einzelne Bits, Ziffern, Worte oder Mehrfach-Worte sein. Häufig werden einzelne Bits und kurze Felder kombiniert und von einem einzelnen Eingangs- und Ausgangsport verarbeitet.

Die Kombination von Status- und Steuer-Informationen in Bytes verringert die gesamte Anzahl der E/A-Port-Adressen, die von den peripheren Bausteinen benötigt werden.

Die Kombination bedeutet jedoch, daß individuelle Status-Eingangs-Bits getrennt interpretiert und Steuer-Ausgangs-Bits getrennt bestimmt werden müssen. Die Verfahren für die Trennung der Status-Bits sind folgende:

TRENNUNG DER STATUS-INFORMATIONEN

Trennung der Status-Bits

Schritt 1) Lies Status-Daten vom peripheren Baustein.
Schritt 2) UNDiere logisch mit einer Maske (die Maske hat Einsen in Bit-Positionen, die geprüft werden müssen, andernfalls Nullen).
Schritt 3) Schiebe die getrennten Bits in die niedrigstwertigen Bit-Positionen.

Schritt 3 ist überflüssig, wenn das Feld aus einem einzelnen Bit besteht, da das Null-Flag das Komplement dieses Bits nach dem Schritt 2 enthalten wird (versuchen Sie es!). Ein Schiebe- oder Ladebefehl kann Schritt 2 ersetzen, wenn das Feld ein einzelnes Bit ist und die niedrigstwertige, höchstwertige oder nächsthöchstwertige Bit-Position (Positionen 0, 7 oder 6) belegt. Diese Positionen werden oft für die am häufigsten verwendete Status-Information reserviert. Sie sollten versuchen, die erforderlichen Befehls-Sequenzen für den Prozessor 6502 zu schreiben.

Beachten Sie insbesondere die Verwendung des Befehls Bit-Test. Dieser Befehl führt eine logische UND-Operation zwischen dem Inhalt des Akkumulators und dem Inhalt eines Speicherplatzes aus, bewahrt jedoch das Ergebnis nicht auf. Die Flags werden wie folgt gesetzt:

Null-Flag = 1, wenn das logische UND ein Null-Resultat erzeugt, 0 wenn es nicht der Fall ist.

Negativ-Flag = Bit 7 des Inhaltes des Speicherplatzes (unabhängig vom Wert im Akkumulator).

Überlauf-Bit = Bit 6 des Inhaltes des Speicherplatzes (unabhängig vom Wert im Akkumulator).

KOMBINIEREN VON STEUER-INFORMATIONEN

Setzen und Löschen von Steuer-Bits

Schritt 1) Lies vorhergehende Steuer-Information
Schritt 2) UNDiere logisch mit einer Maske, um Bits zu löschen (die Maske besitzt Nullen in Bit-Positionen die zu löschen sind, anderswo Einsen).
Schritt 3) ODERieren logisch mit einer Maske, um die Bits zu setzen (die Maske besitzt Einsen in Bit-Positionen die zu löschen sind, anderswo Nullen).
Schritt 4) Sende neue Steuer-Information zum peripheren Baustein.

Hier ist das Verfahren wiederum einfacher, wenn das Feld aus einem einzelnen Bit besteht und eine Position an einem von beiden Enden des Wortes belegt.

Einige Beispiele für das Trennen und Kombinieren von Status-Bits sind:

1) Ein 3-Bit-Feld in den Bit-Positionen 2 bis 4 des VIA-Ausgangs-(Daten)-Registers ist ein Eichfaktor. Plaziere diesen Faktor in den Akkumulator.

```
;
;LIES STATUSDATEN VOM EINGANGSPORT
;
        LDA     VIAOR           ;LIES STATUSDATEN
;
;MASKIERE UNERWÜNSCHTE BITS WEG UND VERSCHIEBE ERGEBNIS
;
        AND     #%00011100 ;MASKIERE EICHFAKTOR
        LSR     A               ;VERSCHIEBE ZWEIMAL FÜR NOMINIERUNG
        LSR     A
```

2) Akkumulator enthält ein 2-Bit-Feld, das in die Bit-Positionen 3 und 4 eines VIA-Ausgangs-(Daten)-Registers plaziert werden muß.

```
TEMP = $0040
MASK = %11100111

;
;BRINGE DATEN ZU FELD-POSITIONEN
;
        ASL     A               ;SCHIEBE DATEN ZU BIT-POSITIONEN 3
                                ; UND 4
        ASL     A
        ASL     A
        AND     #%00011000 ;LÖSCHE ANDERE DATENBITS
        STA     TEMP
;
;KOMBINIERE NEUE FELD-POSITIONEN MIT ANDEREN DATEN
;
        LDA     VIOADR
        AND     HMASK           ;LÖSCHE ZU ÄNDERNDES FELD
        ORA     VIORA           ;KOMBINIERE NEUE DATEN MIT ALTEN
        STA     VIORA           ;GIB KOMBINIERTE DATEN AUS
```

DOKUMENTATION DES STATUS- UND STEUER-TRANSFERS

Die Dokumentation ist ein ernstes Problem bei der Handhabung von Steuer- und Status-Informationen. Die Bedeutung von Status-Eingaben oder Steuer-Ausgaben sind selten offensichtlich. Der Programmierer sollte den Zweck der Eingabe- und Ausgabe deutlich mit Kommentaren wie folgt anzeigen "PRÜFE, WENN LESER EINGESCHALTET IST", "WÄHLE GERADE PARITÄT", oder "AKTIVIERE BITGESCHWINDIGKEITS-ZÄHLER". Die logischen und Verschiebe-Befehle wären andernfalls sehr schwierig zu merken, zu verstehen oder fehlerfrei zu machen.

BEISPIELE

Eine nicht-codierte Tastatur

Zweck: Erkennen des Schließen einer Taste von einer nicht-codierten 3 x 3-Tastatur und Plazieren der Nummer der Taste, die gedrückt wurde, in den Akkumulator.

Tastaturen sind im allgemeinen Anordnungen von Schaltern (siehe Bild 11-24). Eine kleine Anzahl von Tasten ist am leichtesten zu handhaben, wenn jede Taste getrennt mit einem Bit eines Eingangsports verbunden wird. Die Anpassung der Tastatur ist dann die gleiche, wie die Anpassung eines Satzes von Schaltern.

MATRIX-TASTATUR

Tastaturen mit mehr als acht Tasten benötigen mehr als einen Eingangsport und daher Multibyte-Operationen. Dies ist besondersunrationell, wenn die Tasten logisch getrennt sind, wie in einem Rechner oder einem Terminal, bei denen der Anwender nur eine Taste zur gleichen Zeit betätigt. Die Anzahl der erforderlichen Eingangsleitungen kann verringert werden, indem die Tasten in Form einer Matrix verbunden werden, wie in Bild 11-25 gezeigt ist. Nun stellt jede Taste eine Potential-Verbindung zwischen einer Zeile und einer Spalte dar. Die Tastatur-Matrix benötigt n x m externe Leitungen, wobei die Anzahl der Zeilen und m die Anzahl der Spalten ist. Im Vergleich hierzu benötigt man n + m externe Leitungen, wenn jede Taste getrennt wäre. Die Tabelle 11-15 vergleicht die Anzahl von Tasten, die von typischen Konfigurationen benötigt werden.

TASTATUR-ABTASTUNG

Ein Programm kann bestimmen, welche Taste gedrückt wurde, indem sie die externen Leitungen von der Matrix verwendet. Das gebräuchliche Verfahren ist eine "Tastatur-Abtastung". Wir legen Zeile 0 an Masse und prüfen die Spalten-Leitungen. Wenn irgendwelche Leitungen an Masse liegen sind, so wurde eine Taste in dieser Zeile betätigt, wodurch eine Verbindung zwischen Zeile und Spalte bewirkt wurde. Wir können bestimmen, welche Taste gedrückt wurde, indem wir feststellen, welche Spalten-Leitung an Masse liegt, d.h., welches Bit des Eingangsports 0 ist. Wenn keine Spalten-Leitung an Masse liegt, so gehen wir zur Zeile 1 weiter und wiederholen die Abtastung. Man sieht, daß man feststellen kann, ob irgendwelche Tasten gedrückt wurden, indem alle Zeilen gleichzeitig an Masse gelegt und die Spalten geprüft werden.
Die Tastatur-Abtastung erfordert, daß die Zeilen-Leitungen zu einem Ausgangsport geführt werden und die Spalten-Leitungen zu einem Eingangsport. Bild 11-20 zeigt die Anordnung. Die CPU kann eine spezielle Zeile an Masse legen, indem sie eine Null in das entsprechende Bit des Ausgangsports plaziert und Einsen in die anderen Bits. Die CPU kann den Zustand der speziellen Spalte bestimmen, indem sie das entsprechende Bit des Eingangsports prüft.

Tabelle 11-15. Vergleich zwischen unabhängigen Verbindungen und Matrix-Verbindungen für Tastaturen.

Tastatur-Größe	Anzahl der Leitungen mit unabhängigen Verbindungen	Anzahl der Leitungen mit Matrix-Verbindungen
3 x 3	9	6
4 x 4	16	8
4 x 6	24	10
5 x 5	25	10
6 x 6	36	12
6 x 8	48	14
8 x 8	64	16

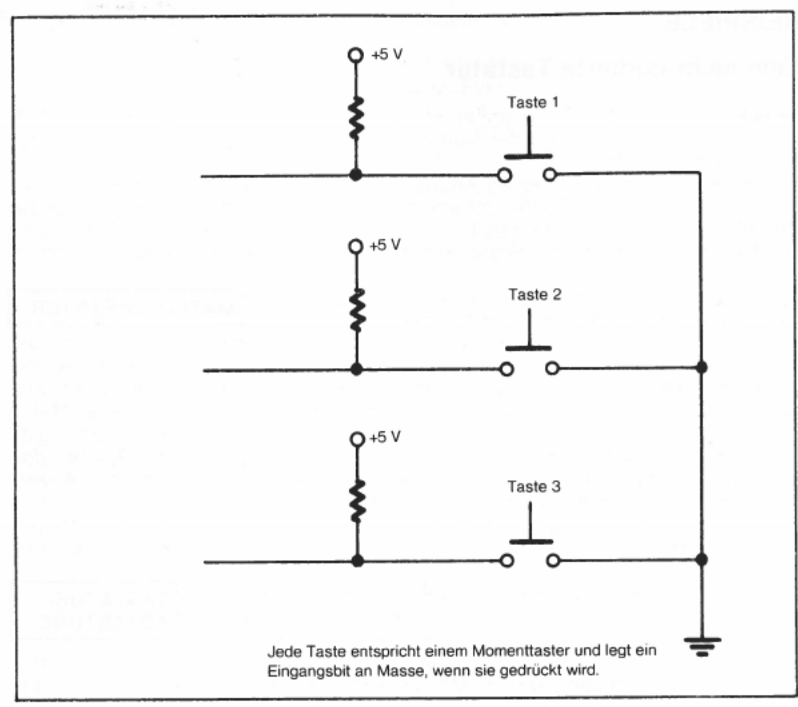


Bild 11-24. Eine kleine Tastatur.

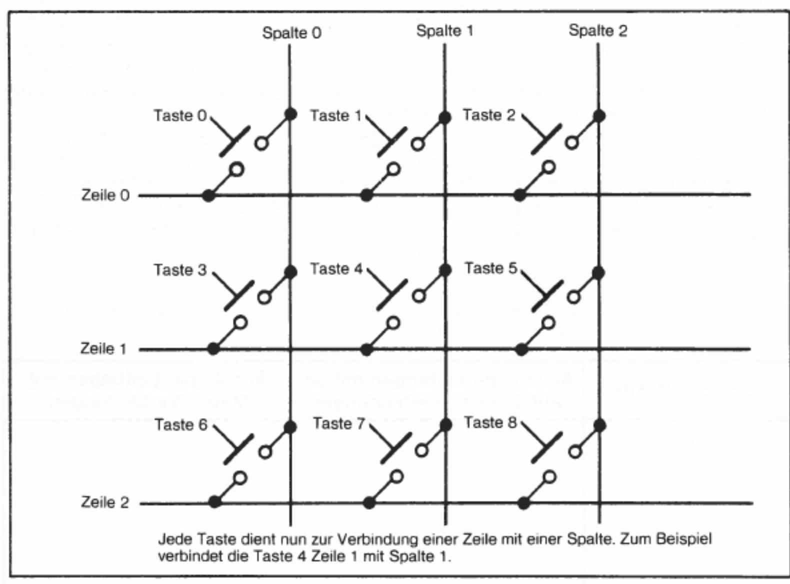


Bild 11-25. Eine Tastatur-Matrix.

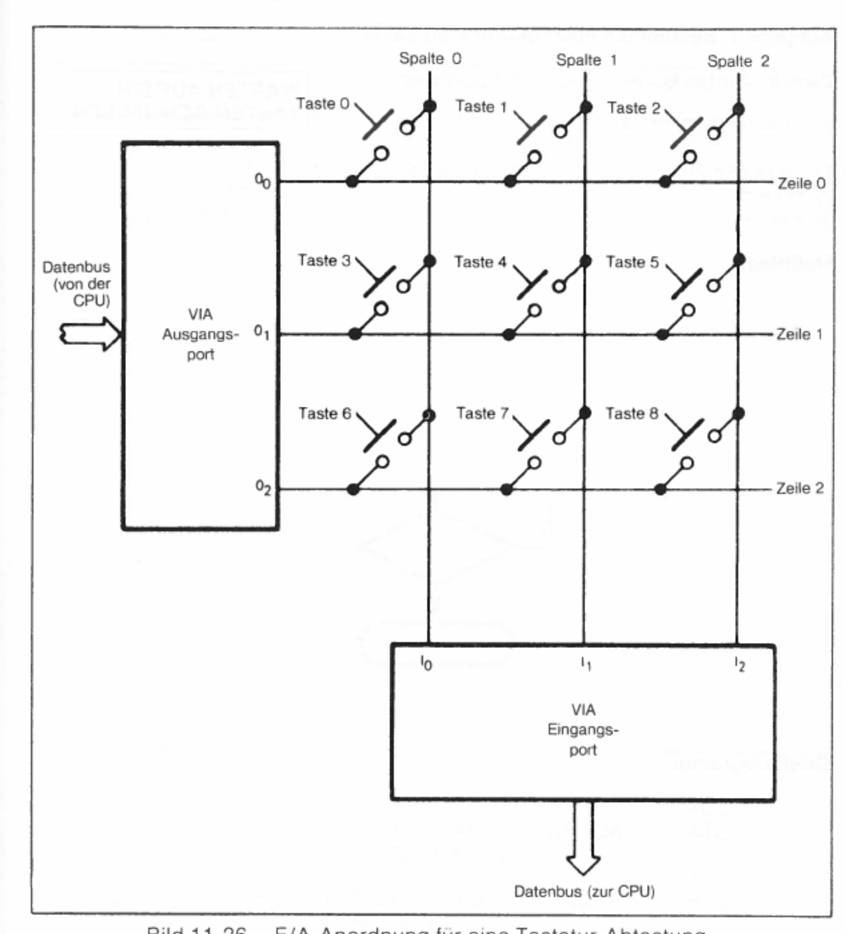


Bild 11-26. E/A-Anordnung für eine Tastatur-Abtastung.

Aufgabe 1: Bestimmen einer gedrückten Taste

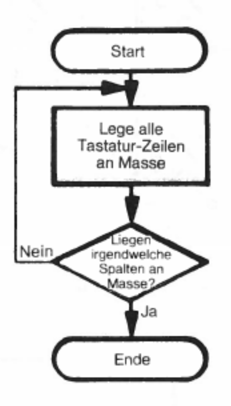
WARTEN AUF EIN TASTEN-SCHLIESSEN

Zweck: Warten bis eine Taste gedrückt wird.

Das Verfahren ist folgendes:

1) Lege alle Zeilen durch Löschen der Ausgangs-Bits an Masse
2) Hole die Spalten-Eingaben durch Lesen des Eingangsports.
3) Kehre zu Schritt 1 zurück, wenn alle Spalten-Eingaben Einsen sind.

Flußdiagramm:



Quellprogramm:

```
        LDA   #$FF
        STA   VIADDRB       ;MACHE PROT-B-LEITUNGEN ZU
                            ;  AUSGÄNGEN
        LDA   #0
        STA   VIAPCR        ;MACHE ALLE STEUER-LEITUNGEN ZU
                            ;  EINGÄNGEN
        STA   VIADDRA       MACHE PORT-A-LEITUNGEN  ZU
                            ;  EINGÄNGEN
        STA   VIAORB        ;LEGE ALLE TASTATUR-ZEILEN AN MASSE
WAITK   LDA   VIAORA        ;HOLE SPALTEN-DATEN DER TASTATUR
        AND   #%00000111    ;MASKIERE SPALTEN-BITS
        CMP   #%00000111    ;LIEGEN IRGENDWELCHE SPALTEN AN
                            ;  MASSE?
        BEQ   WAITK         ;NEIN, WARTE BIS ES BEI EINER DER FALL
                            ;  IST
        BRK
```


Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
0000	A9		LDA	#$FF
0001	FF			
0002	8D		STA	VIADDRB
0003, 0004	VIADDRB			
0005	A9		LDA	#0
0006	00			
0007	8D		STA	VIAPCR
0008, 0009	VIAPCR			
000A	8D		STA	VIADDRA
000B, 000C	VIADDRA			
000D	8D		STA	VIAORB
000E, 000F	VIAORB			
0010	AD	WAITK	LDA	VIAORA
0011, 0012	VIAORA			
0013	29		AND	#%00000111
0014	07			
0015	C9		CMP	#%00000111
0016	07			
0017	F0		BEQ	WAITK
0018	F7			
0019	00		BRK	

Port B des VIA ist der Tastatur-Ausgangsport und Port A ist der Eingangsport.

Das Ausmaskieren aller außer der Spalten-Bits beseitigt jegliche Probleme, die durch die Zustände der unbenützten Eingangsleitungen entstehen könnten.

Wir könnten die Routine verallgemeinern, indem wir den Ausgang und die Maskier-Muster angeben:

```
ALLG    =%11111000
OPEN    =%00000111
```

Diese Bezeichnungen könnten dann im tatsächlichen Programm verwendet werden. Eine andere Tastatur würde nur eine Änderung der Definitionen und eine Neu-Assemblieruung erfordern.

Natürlich ist nur eine Seite des VIA für eine 3 x 3 oder 4 x 4-Tastatur erforderlich. Versuchen Sie das Programm neu zu schreiben, so daß es nur Port A verwendet.

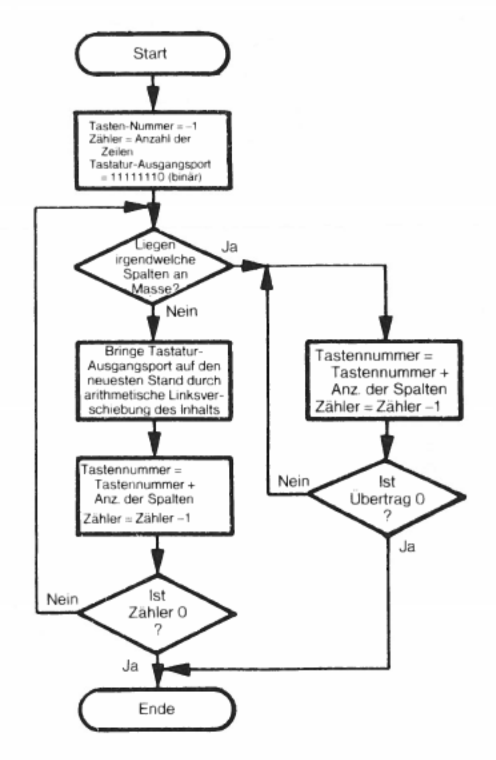
Aufgabe 2: Identifizieren einer Taste

Zweck: Identifizieren einer Tasten-Betätigung durch Plazieren der Nummer der Taste in den Akkumulator.

Das Verfahren sind folgendermaßen aus:

1) Setze Tastennummer auf –1, Tastatur-Ausgangsport auf lauter Einsen mit Ausnahme einer Null in Bit 0, und den Zeilenzähler auf die Anzahl der Zeilen.
2) Hole die Spalten-Eingänge durch Lesen des Eingangsports.
3) Wenn irgendwelche Spalten-Eingänge null sind, gehe zu Schritt 7 weiter.
4) Addiere die Anzahl der Spalten zur Tasten-Nummer, um die nächste Zeile zu erreichen.
5) Bringe den Inhalt des Ausgangsports durch Verschieben des Null-Bits um eine Stelle nach links auf den neuesten Stand.
6) Dekrementiere Zeilenzähler. Gehe zu Schritt 2, falls irgendwelche Zeilen nicht abgetastet wurden, andernfalls gehe zu Schritt 9.
7) Addiere 1 zur Tastennummer. Verschiebe Spalten-Eingaben um ein Bit nach rechts.
8) Wenn Übertrag = 1, kehre zu Schritt 7 zurück.
9) Ende des Programms.

Flußdiagramm:



Quellprogramm:

```
        LDA   #0
        STA   VIAPCR       ;MACHE ALLE STEUER-LEITUNGEN ZU EIN-
                           ;  GÄNGEN
        STA   VIADDRA      ;MACHE PORT-A-LEITUNGEN ZU
                           ;  EINGÄNGEN
        LDA   #$FF
        STA   VIADDRB      ;MACHE POT-B-LEITUNGEN ZU
                           ;  AUSGÄNGEN
        TAX                ;TASTENNUMMER = –1
        LDA   #%11111110   ;STARTE DURCH LEGEN VON ZEILE
                           ;  NULL AN MASSE
        STA   VIAORB
        LDY   #3           ;ZÄHLER IST ANZAHL DERZEILEN
FROW    LDA   VIAORA       ;HOLE SPALTEN-EINGANG B
        AND   #%00000111   ;ISOLIERE SPALTEN-BITS
        CMP   #%00000111   ;LIEGEN IRGENDWELCHE SPALTEN AN
                           ;  MASSE?
        BNE   FCOL         ;JA, BESTIMME WELCHE
        TXA                ;NEIN, BRINGE TASTENNUMMER ZUR
                           ;  NÄCHSTEN ZEILE
        CLC
        ADC   #3           ;DURCH ADDIEREN DER ANZAHL DER
                           ;  SPALTEN
        TAX
        ASL   VIAORB       ;BRINGE ABTAST-MUSTER FÜR NÄCHSTE
                           ;  ZEILE AUF DEN NEUESTEN STAND
        DEY                ;WURDEN ALLE ZEILEN ABGETASTET?
        BNE   FROW         ;NEIN, TASTE NÄCHSTE AB
        BRK
FCOL    INX                ;TASTENNUMMER = TASTENNUMMER + 1
        LSR   A            ;IST DIES DIE AN MASSE LIEGENDE
                           ;  SPALTE?
        BCS   FCOL         ;NEIN,PRÜFE NÄCHSTE
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A9		LDA	#0
0001	00			
0002	8D		STA	VIAPCR
0003 0004	VIAPCR		LD	A,00001111B
0005	8D		STA	VIADDRA
0006 0007	VIADDRA			
0008	A9		LDA	#$FF
0009	FF			
000A	8D		STA	VIADDRB
000B 000C	VIADDRB			
000D	AA		TAX	
000E	A9		LDA	#%11111110
000F	FE			
0010	8D		STA	VIAORB
0011 0012	VIAORB			
0013	A0		LDY	#3
0014	03			
0015	AD	FROW	LDA	VIAORA
0016 0017	VIAORA			
0018	29		AND	#%00000111
0019	07			
001A	C9		CMP	#%00000111
001B	07			
001C	D0		BNE	FCOL
001D	0C			
001E	8A		TXA	
001F	18		CLC	
0020	69		ADC	#3
0021	03			
0022	AA		TAX	
0023	0E		ASL	VIAORB
0024 0025	VIAORB			
0026	88		DEY	
0027	D0		BNE	FROW
0028	EC			
0029	00		BRK	
002A	E8	FCOL	INX	
002B	4A		LSR	A
002C	B0		BCS	FCOL
002D	FC			
002E	00		BRK	

Wir haben eine CLC-Befehl für zusätzliche Deutlichkeit eingefügt, er ist jedoch nicht wirklich erforderlich. Der einzige Fall, in dem der BNE-Befehl keine Verzweigung bewirkt, ist derjenige, bei dem die zwei im CMP verwendeten Operanden gleich sind. In diesem Fall wird das Übertrags-Flag immer gesetzt um anzuzeigen, daß kein "Borgen" erzeugt wurde. So könnten wir die Sequenz

```
        CLC
        ADC     #3              ;DURCH ADDIEREN DER ANZAHL DER
                                ;  SPALTEN
```

Durch den Einzelbefehl

```
        ADC     #2              ;DURCH ADDIEREN DER ANZAHL DER
                                ;  SPALTEN (BEACHTE ÜBERTRAG = 1)
```

ersetzen.

Jedesmal, wenn eine Zeilen-Abtastung keinen Erfolg hat, müssen wir die Anzahl der Spalten zur Tasten-Nummer addieren, um so über die momentane Zeile hinauszukommen (versuchen Sie das Verfahren an der Tastatur in Bild 11-26).

Was ist das Ergebnis des Programmes, wenn keine Tasten gedrückt wurden? Ändern Sie das Programm so, daß die Abtastung in diesem Fall neuerlich gestartet wird. Wir könnten einen zusätzlichen INX-Befehl vor dem ersten BRK einfügen. Was wäre der endgültige Wert im Indexregister X, wenn keine Tasten gedrückt werden? Wäre es anders als der Fall, in dem die höchstnumerierte Taste gedrückt war? Beachten Sie, daß das Nullflag auch zum Unterscheiden des Falles verwendet werden könnte, bei dem keine Tasten gedrückt werden. Können Sie erklären weshalb?

Eine Alternative besteht in der bidirektionalen Möglichkeit des VIA. Das Verfahren wäre:

1) Lege alle Spalten an Masse und bewahre die Zeilen-Eingänge auf.

2) Lege alle Zeilen an Masse und bewahre Spalten-Eingänge auf.

3) Verwende die Zeilen- und Spalten-Eingänge zusammen, um die Tasten-Nummer aus einer Tabelle zu bestimmen.

Versuchen Sie ein Programm zur Ausführung dieses Verfahrens zu schreiben.

Das Programm kann verallgemeinert werden, indem die Nummern der Zeilen, die Nummern der Spalten und die Maskier-Muster in Parameter mit Namen mit EQUATE (=)-Pseudo-Operationen verwandelt werden.

Eine codierte Tastatur[18]

Zweck: Hole die Daten,wenn sie von einer codierten Tastatur verfügbar sind, die einen Impuls zusammen mit jedem Daten-Transfer liefert.

Eine codierte Tastatur liefert einen eindeutigen Code für jede Taste. Sie besitzt interne Elektronik, die die Abtastung und Identifizierung des vorhergehenden Beispiels ausführt. Man hat die Wahl zwischen der einfacheren Software, die von der codierten Tastatur benötigt wird und den niedrigeren Hardware-Kosten der nicht codierten Tastatur.

Codierte Tastaturen können Dioden-Matrizen verwenden, TTL-Codierer oder MOS-Codierer. Die Codes können ASCII, EBCDIC oder ein sonstiger Code sein. Die Codierschaltungen enthalten häufig PROMs.

ROLLOVER

Die Codierschaltung kann mehr ausführen, als nur das Schließen der Tasten codieren. Sie kann auch die Tasten entprellen und das "Rollover", das Problem der Betätigung von mehreren Tasten zur gleichen Zeit, verarbeiten. Bekannte Verfahren für die Handhabung des Rollover beinhalten das "2-Key-Rollover", wobei zwei Tastenbestätigungen (jedoch nicht mehr) zur gleichen Zeit in getrennte Betätigungen aufgelöst werden, sowie das "n-Key-Rollover", wobei jede Anzahl von Tastenbetätigungen zur gleichen Zeit in getrenntes Schließen der Tasten aufgelöst wird.

Eine codierte Tastatur liefert auch einen Impuls mit jedem Datentransfer. Dieser Impuls signalisiert ein neues Schließen einer Taste. Bild 11-27 zeigt das Interface zwischen einer codierten Tastatur und dem Mikroprozessor 6502.

Der Versatile-Interface-Adapter 6522 ermöglicht einen Eingangs-Zwischenspeicher an beiden Ports A und B. Diese Zwischenspeicher werden durch Setzen von Bit 1 (für Port B) oder Bit 0 (für Port A) oder des Hilfs-Steuerregisters (siehe Bild 11-10) freigegeben. In dieser Betriebsart werden die Daten an den Eingangs-Anschlüssen zwischengespeichert, wenn das Unterbrechungsflag gesetzt ist und werden sich nicht ändern, bis das Unterbrechungsflag gelöscht wird. Beachten Sie, daß die Zwischenspeicher etwas anders als die auf der B-Seite arbeiten, bei denen der Inhalt der Ausgangsregister zwischengespeichert wird, wenn der Anschluß als Ausgang programmiert ist.

Der Tastatur-Tastimpuls wird zum Eingang CA1 geführt. Ein Übergang an der Tastleitung bewirkt, daß das Unterbrechungsflag-Registerbit 1 auf High geht. Bit 0 des peripheren Steuerregisters (siehe Bild 11-9) bestimmt, ob der VIA High-auf-Low-Übergänge an CA1 (Bit 0 = 0) oder Low-auf-High-Übergänge (Bit 0 = 1) erkennt.

Der VIA enthält einen seriellen, flanken-empfindlichen zwischengespeicherten Statusport, sowie einen Datenport. Er enthält ferner einen Inverter, der zur Handhabung von Impulsen jeder Polarität verwendet werden kann. Der VIA kann mehrere einfache Schalterelemente ersetzen. Der Entwickler kann Korrekturen durchführen, indem er den Inhalt des Steuer-Registers (in Software) ändert, bevor er einen Aufbau neu entwirft. Zum Beispiel benötigt die Änderung der aktiven Flanke nur die Änderung eines einzelnen Programm-Bits, während es andernfalls zusätzliche Schalt-Elemente und eine neue Verdrahtung erfordern würde.

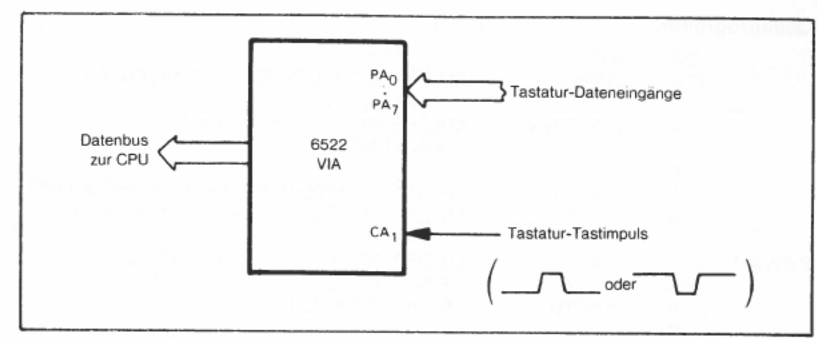
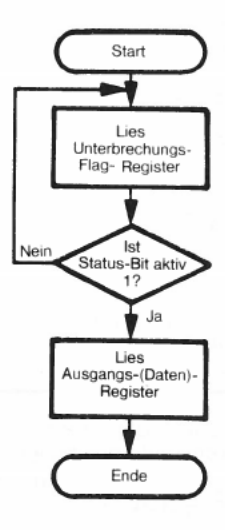


Bild 11-27. E/A-Interface für eine codierte Tastatur.

Aufgabe: Eingabe von Tastatur

Zweck: Warte auf einen Impuls mit aktiv Low vom VIA auf der Steuerleitung CA1 und plaziere dann die Daten vom Port A in den Akkumulator. Beachten Sie, daß das Lesen der Daten vom Ausgangs-(Daten)Register das Status-Bit im Unterbrechungsflag-Register löscht (diese Schaltung ist Teil des VIA 6522).

Flußdiagramm:



Die Hardware muß die Steuerleitung in einem logischen "1"-Zustand während des Resets halten, um zu verhindern, daß die Status-Flags zufällig gesetzt werden. Ein anfängliches Lesen des Daten-(Ausgangs-)Registers in der Start-Routine kann zum Löschen der Status-Flags verwendet werden. Wie bereits früher erwähnt, können Sie auch die Bits im Unterbrechungs-Flag-Register des 6522 durch Einschreiben von Einsen löschen.

Quellprogramm:

```
        LDA   #0
        STA   VIAPCR        ;MACHE ALLE STEUERLEITUNGEN ZU
                            ;  EINGÄNGEN
        STA   VIADDRA       ;MACHE PORT-A-LEITUNGEN ZU
                            ;  EINGÄNGEN
        LDA   #%00000001
        STA   VIAACR        ;GIB ZWISCHENSPEICHER AN PORT A FREI
        LDA   #%00000010    ;MACHE MUSTER FÜR PRÜFUNG DES CA1-
                            ;  FLAGS
KBWAIT  BIT   VIAIFR        ;LIEGEN NEUE TASTATUR-DATEN VOR?
        BPQ   KBWAIT        ;NEIN, WARTE BIS DIES DER FALL IST
        LDA   VIAORA        ;JA, HOLE DATEN
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
0000	A9		LDA	#0
0001	00			
0002	8D		STA	VIAPCR
0003 0004	VIAPCR			
0005	8D		STA	VIADDRA
0006 0007	VIADDRA			
0008	A9		LDA	#%00000001
0009	01			
000A	8D		STA	VIAACR
000B 000C	VIAACR			
000D	A9		LDA	#%00000010
000E	02			
000F	2C	KBWAIT	BIT	VIAIFR
0010 0011	VIAIFR			
0012	F0		BEQ	KBWAIT
0013	FB			
0014	AD		LDA	VIAORA
0015 0016	VIAORA			
0017	00		BRK	

Um zu veranlassen, daß das Status-Bit auf einen Low-auf-High-Übergang auf CA1 reagiert, müssen Sie Bit 0 des Unterbrechungs-Flag-Registers setzen.

Die anderen Quittierungs-Statusflags sind Bit 0 (für CA2), 3 (für CB2) und 4 (für CB1) des Unterbrechungsflag-Registers.

Zeigen Sie, daß das Lesen des Ausgangs-(Daten)-Registers das Status-Bit löscht. Hinweis: Retten Sie den Inhalt des Unterbrechungs-Flag-Registers in den Speicher, bevor der Befehl LDA VIAORA ausgeführt wird. Was geschieht, wenn man LDA durch STA ersetzt? Wie ist es mit CMP, INC, ROL?

Beachten Sie, daß entweder das Lesen oder Schreiben des Ausgangs-(Daten)-Registers das Statusbit löscht. Was geschieht, wenn Sie Port A von der Adresse ohne Quittierung lesen (Tabelle 11-7)? Was geschieht, wenn Sie LDA VIAORA durch LDA VIAORB ersetzen?

Ein Digital-Analog-Wandler[19-22]

Zweck: Sende Daten zu einem 8-Bit-Digital-Analog-Wandler, der eine Zwischenspeicher-Freigabe mit aktiv-Low besitzt.

Digital-Analog-Wandler erzeugen ein kontinuierliches Signal, das von Motoren, Heizwicklungen, Relais und anderen elektrischen und mechanischen Ausgangs-Bausteinen benötigt wird. Typische Wandler bestehen aus Schaltern und Widerstands-Ketten mit den entsprechenden Widerstands-Werten. In der Regel muß man eine Referenz-Spannung vorsehen, sowie weitere digitale und analoge Schaltungen, obwohl vollständige Einheiten nunmehr preisgünstig zur Verfügung stehen.

Bild 11-28 beschreibt den 8-Bit-D/A-Wandler Signetics NE5018, der auf dem Chip einen parallelen Dateneingangs-Zwischenspeicher mit 8 Bits enthält. Ein niedriger Pegel auf dem $\overline{LE}$-Eingang (Latch Enable) taktet die Eingangsdaten in den Zwischenspeicher, wo sie verbleiben, nachdem $\overline{LE}$ wieder auf High gegangen ist.

D/A-WANDLER INTERFACE

Bild 11-29 zeigt die Anpassung des Bausteins an ein 6502-System. Beachten Sie, daß die B-Seite des VIA automatisch einen Impuls mit aktiv Low erzeugt, der zum Einspeichern der Daten in den Wandler erforderlich ist. CB2 dient als ein Signal für "Ausgang Bereit" (Output Ready). Erinnern Sie sich daran, daß CB2 automatisch für die Dauer eines Zyklus auf Low geht, nach einer Schreib-Operation beim Datenregister am B-Port des Ausgangs-(Daten)-Registers, wenn CB2 in der Impuls-Ausgangs-Betriebsart ist, (siehe Tabelle 11-59). Die peripheren Steuerregister-Bits sind:

Bit 7 = 1, um CB2 zu einem Ausgang zu machen
Bit 6 = 0, um CB2 zu einem Impuls zu machen
Bit 5 = 1, um CB2 zu einem kurzen "Output Ready"-Impuls zu machen (für die Dauer eines Takt-Zyklus).

Beachten Sie, daß der VIA einen Ausgabe-Zwischenspeicher enthält. Die Daten bleiben daher während und nach der Umwandlung stabil. Der Wandler benötigt typisch einige wenige Mikrosekunden, um ein analoges Ausgangssignal zu erzeugen. Daher kann der Zwischenspeicher des Wandlers freigegeben bleiben, wenn der Port nicht für andere Zwecke verwendet wird.

In Anwendungen, in denen 8 Bit Auflösung nicht ausreichen, können 10- bis 16-Bit-Wandler verwendet werden. Eine zusätzliche Port-Logik ist erforderlich, um alle Datenbits zu verarbeiten. Einige Wandler liefern einen Teil dieser Logik.

Der VIA dient sowohl als paralleler Daten-Port, wie auch als Steuerport. CB2 ist ein Impuls, der einen Takt-Zyklus nach dem Einspeichern der Daten in den VIA dauert. Dieser Impuls ist lang genug, um die Erfordernisse des Konverters NE5018 zu erfüllen (typisch 400 ns).

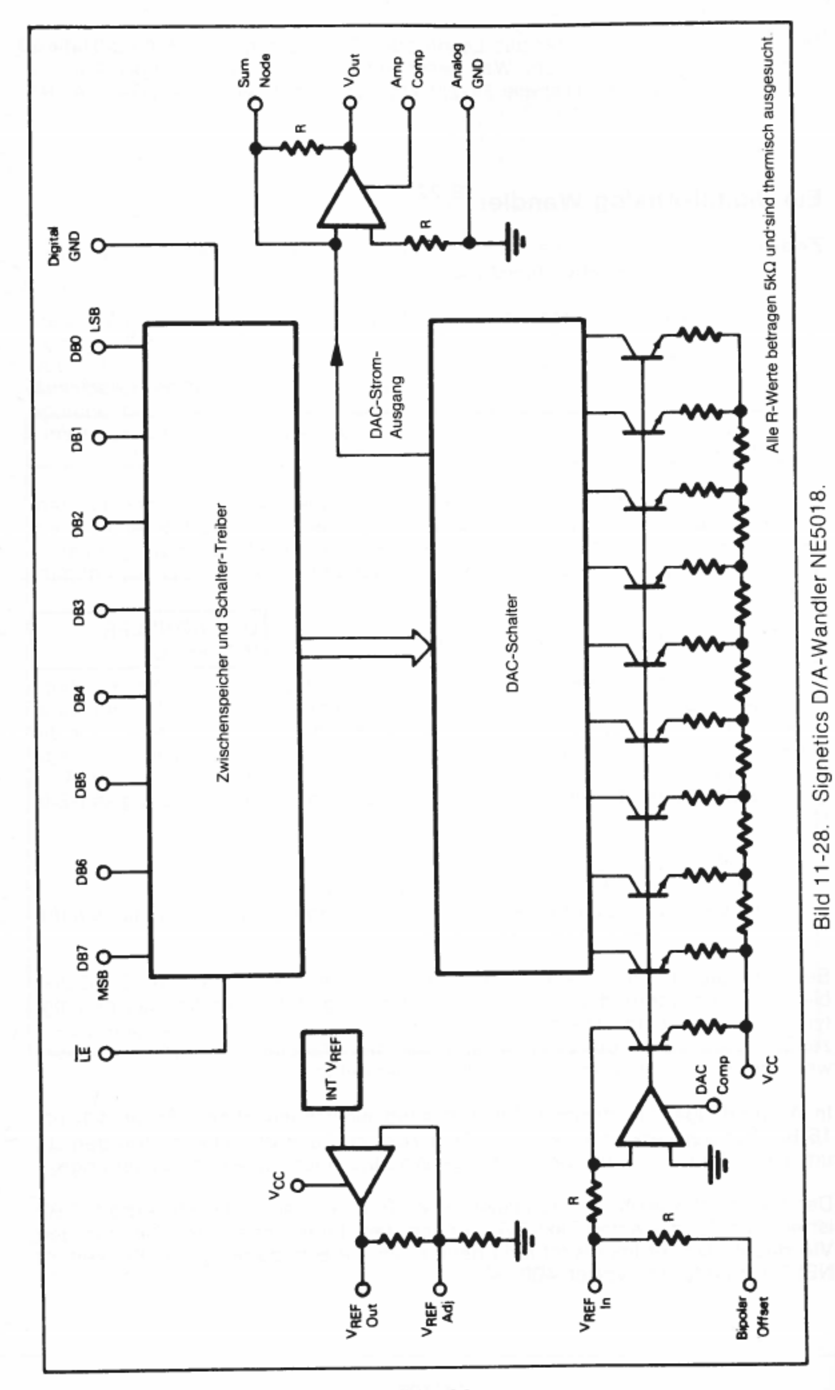


Bild 11-28. Signetics D/A-Wandler NE5018.

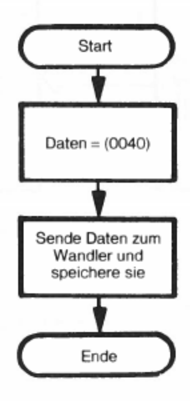
Datenbus von der CPU
6522 VIA
PB_7
PB_0
CB_2
NE5018 D/A-Wandler
$\overline{LE}$
Analog-Ausgang

Bild 11-29. Interface für einen 8-Bit-Digital/Analog-Wandler.

Aufgabe: Ausgabe zum Konverter

Zweck: Sende Daten vom Speicherplatz 0040 zum Wandler.

Flußdiagramm:



Quellprogramm:

```
        LDA   #$FF
        STA   VIADDRB     ;MACHE PORT-B-LEITUNGEN ZU
                          ;  AUSGÄNGEN
        LDA   #%10100000
        STA   VIAPCR      ;LIEFERE KURZEN
                          ;  ZWISCHENSPEICHER-FREIGABE-IMPULS
        LDA   $40         ;HOLE DATEN
        STA   VIAORB      ;SENDE DATEN ZUM DA-WANDLER UND
                          ;  ZWISCHENSPEICHER
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)	
0000	A9	LDA	#$FF
0001	FF		
0002	8D	STA	VIADDRB
0003 } 0004 }	VIADDRB		
0005	A9	LDA	#%10100000
0006	A0		
0007	8D	STA	VIAPCR
0008 } C009 }	VIAPCR		
000A	A5	LDA	$40
000B	40		
000C	8D	STA	VIAORB
000D } 000E }	VIAORB		
000F	00	BRK	

Der Impuls für den Zwischenspeicher-Freigabe-Eingang wird automatisch erzeugt, wenn Daten in das Ausgangs (Daten)-Register B gespeichert werden. Beachten Sie, daß der Impuls ziemlich kurz ist und nur einen Taktzyklus dauert. Dies könnte für einige Anwendungen nicht ausreichen.

Wir könnten den (manuellen) Pegel-Ausgang von CB2 verwenden, wenn das Zwischenspeicher-Freigabesignal aktiv High wäre, oder wenn die erforderliche Länge größer wäre. Das Programm würde dann folgendermaßen aussehen:

```
        LDA   #$FF
        STA   VIADDRB     ;MACHE PORT-B-LEITUNGEN ZU
                          ;  AUSGÄNGEN
        LDA   #%11000000
        STA   VIAPCR      ;MACHE ZWISCHENSPEICHER-FREIGABE
                          ;  ZU EINEM PEGEL (LOW)
        LDA   $40         ;HOLE DATEN
        STA   VIAORB      ;SENDE DATEN ZUM
                          ;  DA-WANDLER-AUSGANGSPORT
        LDA   #%11100000
        STA   VIAPCR      ;ÖFFNE DA-WANDLER-ZWISCHENSPEICHER
                          ;  (FREIGABE HIGH)
        LDA   #%11000000
        STA   VIAPCR      ;SPEICHERE DATEN (FREIGABE LOW)
        BRK
```

Hier wird Bit 6 des peripheren Steuerregisters gesetzt, um CB2 zu einem Pegel zu machen, mit einem Wert, der durch Bit 5 des peripheren Steuerregisters gegeben ist. Dies wird als die "manuelle" Ausgangs-Betriebsart in der Literatur des 6522 bezeichnet. Beachten Sie, wieviel mehr Befehle erforderlich sind, um die Zwischenspeicher-Freigabe (Latch Enable) wie im vorhergehenden Beispiel zu pulsen, da kein automatischer Impuls geliefert wird. Ein invertierendes Gatter könnte auch zum Invertieren der Polarität des Tastimpulses verwendet werden.

In der manuellen Betriebsart ist CB2 vollständig unabhängig vom parallelen Datenport. Er ist einfach ein Steuerausgang, der für jeden Zweck verfügbar ist. Das einzige Problem bei der Verwendung besteht darin, daß Sie nicht zufällig irgendeines der anderen Bits im peripheren Steuerregister ändern dürfen, da Sie sonst unkontrollierte Funktionen erzeugen.

Analog-Digital-Wandler[19-23]

Zweck: Hole Daten von einem Analog-Digital-Wandler mit 8 Bits, der einen Impuls "Starte Umwandlung" (Start Conversion) benötigt, um den Umwandlungsprozess in Gang zu setzen und eine Leitung "End of Conversion" (Ende der Umwandlung) besitzt um den Abschluß des Vorganges anzuzeigen, sowie die Verfügbarkeit gültiger Daten.

Analog-Digital-Wandler verarbeiten das kontinuierliche Signal, das von verschiedenen Arten von Sensoren und Wandlern erzeugt wird. Der Wandler erzeugt das digitale Ausgangssignal, das vom Computer benötigt wird.

Eine Form eines Analog-Digital-Wandlers ist der Baustein für die sukzessive Approximation, der einen direkten 1-Bit-Vergleich während jedes Takt-Zyklus ausführt. Derartige Wandler sind schnell, besitzen jedoch eine geringe Rausch-Unempfindlichkeit. Die integrierenden Dual-Slope-Konverter sind eine weitere Art der Analog-Digital-Wandler. Diese Bausteine benötigen mehr Zeit, sind jedoch nicht so empfindlich gegen Rauschen. Es werden auch andere Verfahren, wie etwa die schrittweise Ladungs-Kompensation verwendet.

Analog-Digital-Wandler benötigen gewöhnlich einige externe, analoge und digitale Schaltungen. Vollständige Einheiten sind nach und nach preisgünstiger erhältlich.

Bild 11-30 zeigt den A/D-Wandler mit 8 Bits MM5357 von National. Der Baustein erhält einen Zwischenspeicher für das Ausgangssignal und Tristate-Datenausgänge. Ein Impuls auf der Leitung "Start Conversion" (STRT CONV) startet die Umwandlung des Analog-Einganges.

Nach etwa 40 Taktzyklen (der Wandler benötigt einen Takt mit TTL-Pegel mit einer Mindestimpulsbreite von 400 ns), wird das Ergebnis zu den Ausgangs-Zwischenspeicher gelangen und der Ausgang "End of Conversion" (EOC) wird dies anzeigen, indem er auf High geht. Daten werden von den Zwischenspeichern gelesen, indem eine 1 zum Eingang "Ausgangs-Freigabe" geführt wird. Bild 11-31 zeigt die Verbindung für den Baustein und einige typische Anwenderschaltungen.

A/D-WANDLER-INTERFACE

Bild 11-32 zeigt das Interface für den Prozessor 6502 und den A/D-Wandler 5357. Die Steuerleitung CA2 wird in der manuellen (Pegel)-Ausgabe-Betriebsart verwendet, um einen Impuls "Start Conversion" (aktiv-High) ausreichender Länge zu liefern. Das Signal "End of Conversion" wird zur Steuerleitung CA1 geführt, so daß, wenn EOC auf High geht, das Bit 1 des Unterbrechungsflag-Registers gesetzt wird. Die wichtige Flanke auf der Leitung "End of Conversion" ist die Flanke Low-auf-High, die den Abschluß der Umwandlung anzeigt. Beachten Sie, daß wir bei der Verwendung des Bausteins 6522 sowohl den Steuereingang als auch den Steuerausgang handhaben, da das Wandler-Interface eine vollständige Quittierung beinhaltet. Der Anschluß "Output Enable" am Konverter wird auf High gelegt, da wir die Daten nicht direkt auf den Tristate-Datenbus des Prozessors plazieren. Beachten Sie (siehe Bild 11-30), daß die Datenausgänge des Konverters komplementierte Binärwerte sind (alles Nullen sind Vollauschlag).

NATIONAL
MM 5357 8-Bit-A/D-Wandler

Allgemeine Beschreibung

Der MM5357 ist ein monolithischer 8-Bit-A/D-Wandler, der in ionen-implantierter P-Kanal-MOS-Technologie gefertigt wird. Er enthält einen Komparator mit hoher Eingangs-Impedanz, 256 Serien-Widerstände und Analog-Schalter, Steuer-Logik und Ausgangs-Zwischenspeicher. Die Umwandlung basiert auf dem Verfahren der sukzessiven Approximation, bei dem die unbekannte Analog-Spannung mit den Verbindungspunkten der Widerstände unter Verwendung von Analog-Schaltern verglichen wird. Wenn die Spannung am entsprechenden Verbindungspunkt mit der unbekannten Spannung übereinstimmt, ist die Umwandlung abgeschlossen und die digitalen Ausgänge enthalten ein komplementäres 8-Bit-Wort, das der unbekannten Spannung entspricht. Die Binär-Ausgänge sind Tristate-Ausgänge, wodurch eine Verbindung mit gemeinsamen Daten-Leitungen möglich ist.

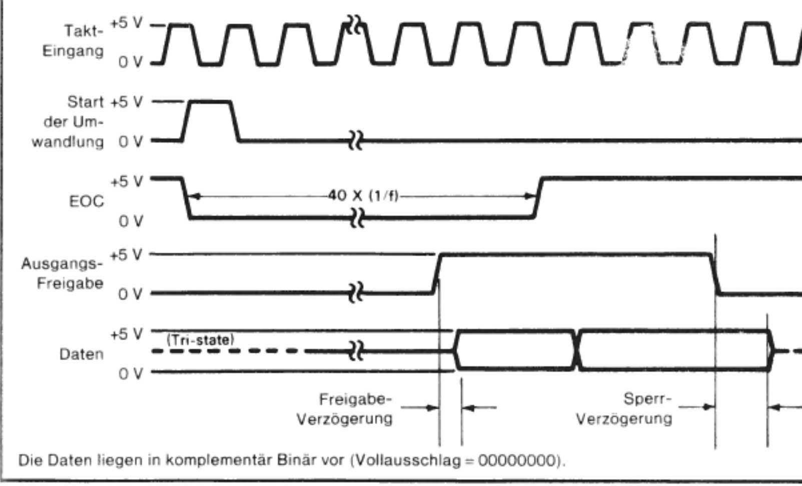
Eigenschaften:

- Niedrige Kosten
- Eingangs-Bereiche ±5 V, 10 V
- Hohe Eingangs-Impedenz
- Tristate-Ausgänge
- Enthält Ausgangs-Zwischenspeicher
- TTL-kompatibel

Wichtigste Spezifikationen

■ Auflösung	8 Bits
■ Linearität	±1/2 LSB
■ Umwandlungs-Geschwindigkeit	40 µs
■ Eingangs-Impedanz	> 100 MΩ
■ Betriebs-Spannungen	+5 V, −12 V, Masse
■ Takt-Bereich	5.0 kHz bis 2.0 MHz

Zeitlicher Ablauf:



Die Daten liegen in komplementär Binär vor (Vollausschlag = 00000000).

Bild 11-30. Allgemeine Beschreibung und Arbeitsweise des A/D-Wandlers National 5357.

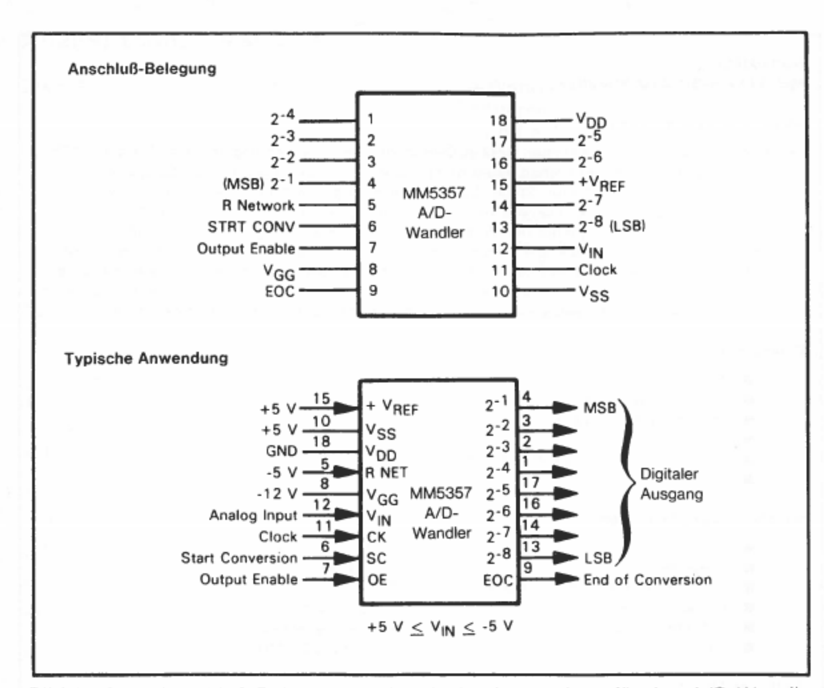


Bild 11-31. Anschluß-Belegung und typische Anwendung für den A/D-Wandler National 5357.

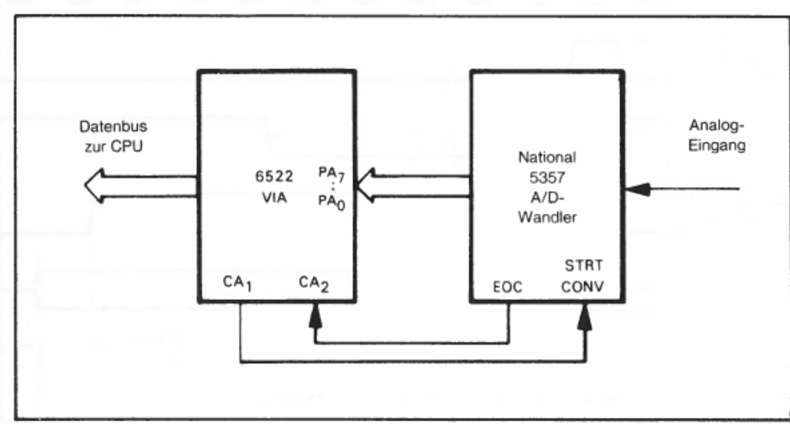
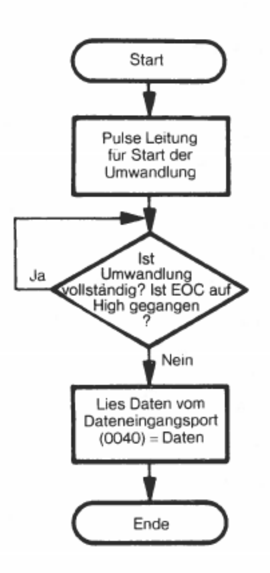


Bild 11-32. Interface für einen 8-Bit-Analog/Digital-Wandler.

Aufgabe: Eingabe vom Wandler

Zweck: Starte den Umwandlungsprozess, warte auf "End of Conversion", das auf Low und dann auf High geht, lies dann die Daten und speichere sie in den Speicherplatz 0040.

Flußdiagramm:



Beachten Sie, daß hier der VIA als ein paralleler Datenport, als Status-Port und als Steuer-Port dient.

Quellprogramm:

```
        LDA   #0
        STA   VIADDRA      ;MACHE PORT-A-LEITUNGEN ZU
                           ;  EINGÄNGEN
        LDA   #%00001101
        STA   VIAPCR       ;BRINGE "START CONV." AUF LOW, GIB
                           ;  EOC LOW-AUF-HIGH FREI
        LDA   #%00001111
        STA   PIACRA       ;PULSE "START VONVERSION" HIGH
        LDA   #%00001101
        STA   VIAPCR       ;PULSE "START CONVERSION" LOW
WTEOC   LDA   VIAIFR
        AND   #%00000010   ;IST UMWANDLUNG BEENDET?
        BNE   WTEOC        ;NEIN, WARTE
        LDA   VIAORA       ;JA, HOLE DATEN VON WANDLER
        EOR   #%11111111   ;KOMPLEMENTIERE DATEN FÜR WAHREN
                           ;  WERT
        STA   $40          ;BEWAHRE KONVERTER-DATEN AUF
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A9		LDA	#0
0001	00			
0002	8D		STA	VIADDRA
0003 0004	VIADDRA			
0005	A9		LDA	#%00001101
0006	0D			
0007	8D		STA	VIAPCR
0008 0009	VIAPCR			
000A	A9		LDA	#%00001111
000B	0F			
000C	8D		STA	VIAPCR
000D 000E	VIAPCR			
000F	A9		LDA	#%00001101
0010	0D			
0011	8D		STA	VIAPCR
0012 0013	VIAPCR			
0014	AD	WTEOC	LDA	VIAIFR
0015 0016	VIAIFR			
0017	29		AND	#%00000010
0018	02			
0019	D0		BNE	WTEOC
001A	F9			
001B	AD		LDA	VIAORA
001C 001D	VIAORA			
001E	49		EOR	#%11111111
001F	FF			
0020	85		STA	$40
0021	40			
0022	00		BRK	

Die VIA-Steuerregister-Bits sind:

Bit 3 = 1, um CA2 zu einem Ausgang zu machen
Bit 2 = 1, um CA2 zu einem Pegel zu machen (Manuelle Ausgabe-Betriebsart)
Bit 1 = Wert des Pegels an CA2
Bit 0 = 1, um das Status-Flag bei einem Low-auf-High-Übergang auf CA1 zu setzen.

Beachten Sie, daß VIAs unter Verwendung der nach-indizierten Adressierung adressiert werden könne. Die Start-Adresse des VIA (VIAORB) wird in zwei Speicherplätzen auf Seite Null plaziert. Alle VIA-Register können mit entsprechenden Versetzungen im Indexregister Y erreicht werden.

Ein Fernschreiber (TTY = Teletypewriter)

TTY-INTERFACE

Zweck: Transferiere Daten zu und von einem seriellen Standard-Fernschreiber mit 10 Zeichen pro Sekunde.

Im allgemeinen transferieren Fernschreiber Daten in einem asynchronen seriellen Bertrieb. Das Verfahren ist folgendes:

STANDARD-TTY ZEICHEN-FORMAT

1) Die Leitung ist normalerweise im Ein-Zustand.
2) Ein Start-Bit (Null-Bit) geht jedem Zeichen voraus.
3) Das Zeichen liegt gewöhnlich im 7-Bit-ASCII vor, wobei das niedrigstwertige Bit zuerst gesendet wird.
4) Das höchstwertige Bit ist ein Paritäts-Bit, das gerade, ungerade oder fest auf Eins oder Null liegen kann.
5) Zwei Stop-Bits (logisch Eins) folgen jedem Zeichen.

Bild 11-33 zeigt das Format. Beachten Sie, daß jedes Zeichen die Übertragung von elf Bits erfordert, von denen nur sieben Informationen enthalten. Da die Daten-Geschwindigkeit 10 Zeichen pro Sekunde beträgt, ist die Bit-Rate gleich 10 x 11 oder 110 Baud. Jedes Bit besitzt daher eine Breite von 1/110 einer Sekunde, oder 9.1 Millisekunden. Diese Breite ist ein Durchschnitt. Die Fernschreiber übertragen nicht mit beliebiger hoher Genauigkeit.

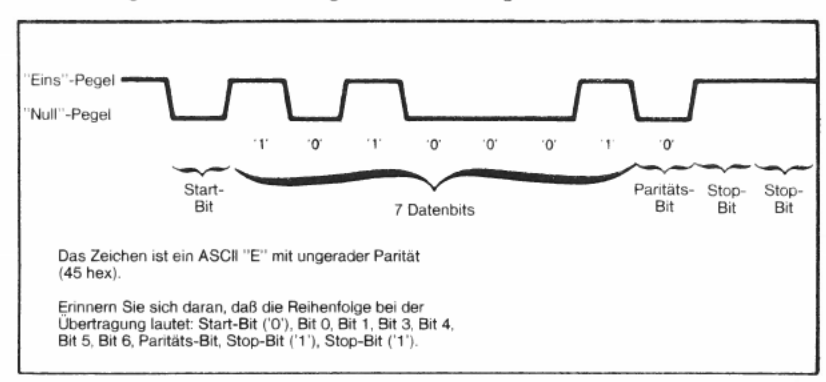


Bild 11-33. Daten eines Fernschreibers.

Damit ein Fernschreiber ordnungsgemäß mit einem Computer kommuniziert, sind folgende Verfahren erforderlich:

Empfang (Flußdiagramm in Bild 11-34):

TTY-EMPFANGS-BETRIEB

Schritt 1) Halte nach einem Start-Bit (einer logischen Null) auf der Datenleitung Ausschau.

Schritt 2) Zentriere den Empfang durch Warten für die Dauer eines halben Bits, oder 4.25 Millisekunden.

Schritt 3) Hole die Datenbits, wobei für die Zeit eines Bits für jedes einzelne gewartet wird. Fasse die Datenbits in ein Wort zusammen, indem zuerst das Bit zum Übertrag geschoben wird und dann die Daten mit dem Übertrag zirkular verschoben werden. Erinnern Sie sich daran, daß das niedrigstwertige Bit zuerst empfangen wird.

Schritt 4) Erzeugen Sie die empfangene Parität und prüfen Sie diese gegenüber der gesendeten Parität. Wenn keine Übereinstimmung vorhanden ist, zeige einen "Paritätsfehler" an.

Schritt 5) Hole die Stop-Bits (warten für die Zeit eines Bits zwischen den Eingängen). Wenn sie nicht korrekt sind (wenn beide Stop-Bits nicht Eins sind), zeige einen "Rahmungsfehler" an.

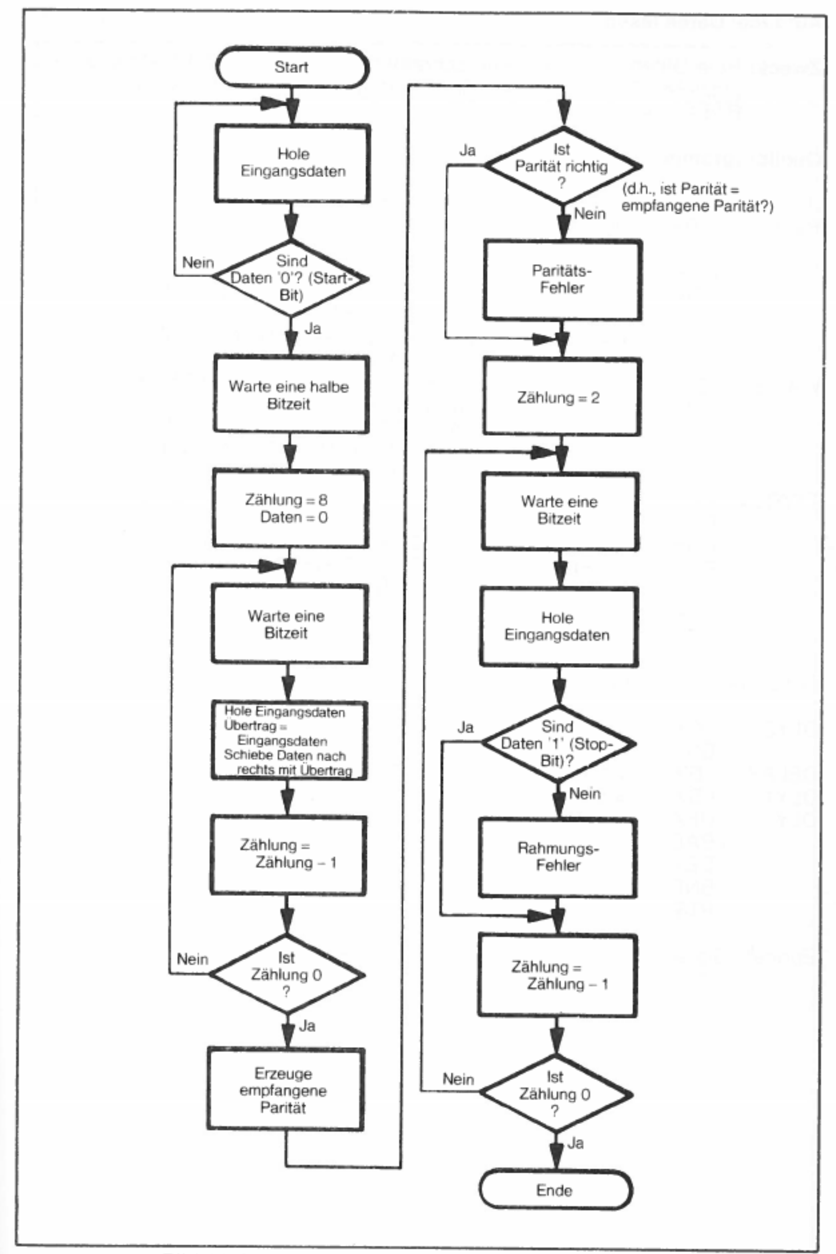


Bild 11-34. Flußdiagramm für Empfangs-Verfahren.

Aufgabe: Daten lesen

Zweck: Hole Daten von einem Fernschreiber über Bit 7 des VIA-Datenports und plaziere die Daten in den Speicherplatz 0046. Für das Verfahren siehe Bild 11-34.

Quellprogramm:

(Es werde angenommen, daß der serielle Port Bit 7 des VIA ist und daß keine Parität oder Rahmung erforderlich ist).

```
        LDA   #0
        STA   VIAPCR        ;MACHE ALLE STEUER-LEITUNGEN ZU
                            ;  EINGÄNGEN
        STA   VIADDRA       ;MACHE PORT-A-LEITUNGEN ZU
                            ;  EINGÄNGEN
WAITS   LDA   VIAORA        ;IST EIN STARTBIT VORHANDEN?
        BMI   WAITS         ;NEIN, WARTE
        JSR   DLY2          ;JA, VERZÖGERE FÜR DIE ZEIT EINES
                            ;  HALBEN BITS ZUR ZENTRIERUNG
        LDA   #%10000000    ;ZÄHLE MIT BIT IM MSB
TTYRCV  JSR   DELAY         ;WARTE FÜR DIE ZEIT EINES BITS
        ROL   PIADRA        ;HOLE DATENBIT
        ROR   A             ;ADDIERE ZUM DATENWORT
        BCC   TTYRCV        ;SETZE FORT, WENN ZÄHLBIT NICHT IM
                            ;  ÜBERTRAG
        STA   $60
        BRK
```

(Verzögerungsprogramm)

```
DLY2    LDY   #5            ;ZÄHLUNG FÜR 4.55 MS
        BNE   DLY1
DELAY   LDY   #10           ;ZÄHLUNG  FÜR 9.1 MS
DLY1    LDX   #$B4          ;HOLE ZÄHLUNG FÜR 0.91 MS
DLY     DEX
        BAE   DLY
        DEY
        BNE   DLY1
        RTS
```

Erinnern Sie sich daran, daß Bit 0 der Daten zuerst empfangen werden.

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A9		LDA	#0
0001	00			
0002	8D		STA	VIAPCR
0003, 0004	VIAPCR			
0005	8D		STA	VIADDRA
0006, 0007	VIADDRA			
0008	AD	WAITS	LDA	VIAORA
0009, 000A	VIAORA			
000B	30		BMI	WAITS
000C	FB			
000D	20		JSR	DLY2
000E	30			
000F	00			
0010	A9		LDA	#%10000000
0011	80			
0012	20	TTYRCV	JSR	DELAY
0013	34			
0014	00			
0015	2E		ROL	VIAORA
0016, 0017	VIAORA			
0018	6A		ROR	A
0019	90		BCC	TTYRCV
001A	F7			
001B	85		STA	$60
001C	60			
001D	00		BRK	
•				
•				
•				
0030	A0	DLY2	LDY	#5
0031	05			
0032	D0		BNE	DLY1
0033	02			
0034	A0	DELAY	LDY	#10
0035	0A			
0036	A2	DLY1	LDX	#$B4
0037	B4			
0038	CA	DLY	DEX	
0039	D0		BNE	DLY
003A	FD			
003B	88		DEY	
003C	D0		BNE	DLY1
003D	F8			
003E	60		RTS	

Dieses Programm nimmt an, daß der Stapel für Unterprogramm-Aufrufe verwendet werden kann, d.h. daß der Monitor den Stapelzeiger bereits initialisiert hat. Andernfalls muß man den Stapelzeiger so initialisieren, wie in Kapitel 10 gezeigt wurde.

Die Konstanten für die Verzögerungs-Routine wurden berechnet, wie früher in diesem Kapitel gezeigt wurde. Sie können versuchen sie selbst zu bestimmen. Die Verzögerungen müssen nicht zu genau sein, da der Empfang zentriert ist, die Nachrichten kurz sind, die Datenrate niedrig ist und der Fernschreiber selbst keine sehr hohe Genauigkeit besitzt.

Wie würden sie dieses Programm erweitern, um die Parität zu prüfen?

Aufgabe 2: Schreiben von Daten

Zweck: Sende Daten zu einem Fernschreiber über Bit 0 eines VIA-Ausgangs (Daten)-Register. Die Daten liegen im Speicherplatz 0060.

TTY-SENDE-BETRIEB

Senden (Flußdiagramm in Bild 11-35):

Schritt 1) Sende ein Start-Bit (d.h. eine logische Eins).

Schritt 2) Sende die sieben Datenbits, beginnend mit dem niedrigstwertigen Bit.

Schritt 3) Erzeuge und sende das Paritäts-Bit.

Schritt 4) Sende zwei Stop-Bits (d.h. logische Einsen).

Die Sende-Routine muß für die Zeit eines Bits zwischen jeder Operation warten.

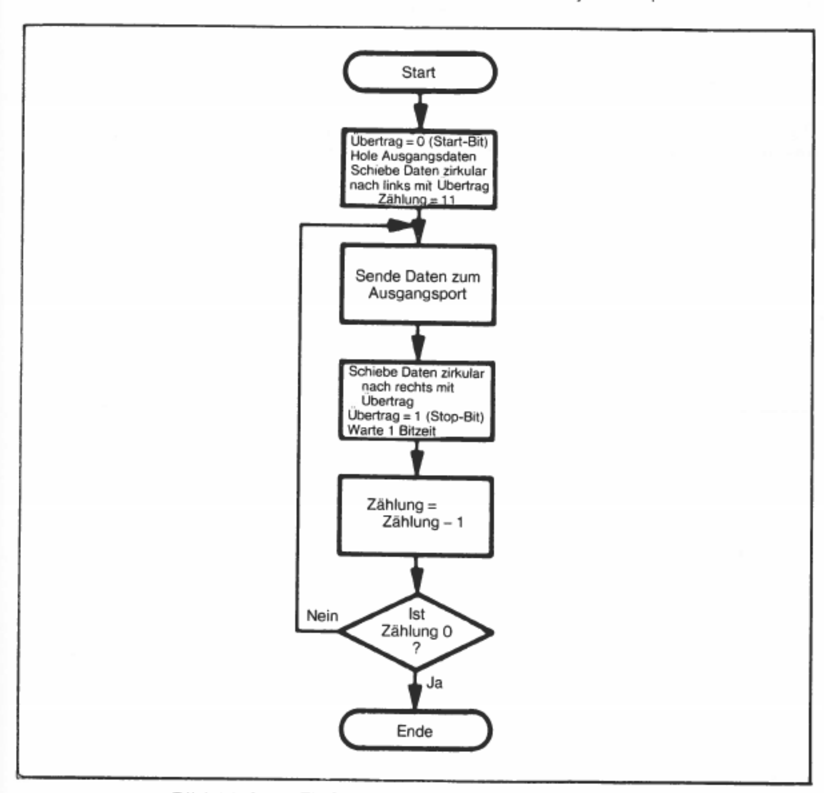


Bild 11-35. Flußdiagramm für Sende-Verfahren.

Quellprogramm:

(Es werde angenommen, daß keine Parität erzeugt werden muß).

```
        LDA   #0
        STA   VIAPCR     ;MACHE ALLE STEUER-LEITUNGEN ZU
                         ;  AUSGÄNGEN
        STA   VIAORB     ;BILDE START-BIT
        LDA   #$FF
        STA   VIADDRB    ;MACHE PORT-B-LEITUNGEN ZU
                         ;  AUSGÄNGEN
        LDA   $60        ;HOLE DATEN
        LDX   #11        ;ZÄHLUNG = 11 BITS IM ZEICHEN
TBIT    JSR   DELAY      ;WARTE FÜR DIE ZEIT EINES BITS
        SEC              ;SETZE ÜBERTRAG ZUR BILDUNG EINES
                         ;  STOP-BITS
        ROR   A          ;HOLE NÄCHSTES BIT DES ZEICHENS
        ROL   VIAORB     ;SENDE NÄCHSTES BIT ZU TTY
        DEX
        BNE   TBIT
        BRK
```

Das hier verwendete DELAY-Unterprogramm muß den Akkumulator und das Index-Register X aufbewahren. Erinnern Sie sich daran, daß Bit 0 der Daten zuerst gesendet werden muß.

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A9		LDA	#0
0001	00			
0002	8D		STA	VIAPCR
0003 0004	VIAPCR			
0005	8D		STA	VIAORB
0006 0007	VIAORB			
0008	A9		LDA	#$FF
0009	FF			
000A	8D		STA	VIADDRB
000B 000C	VIADDRB			
000D	A5		LDA	$60
000E	60			
000F	A2		LDX	#11
0010	0B			
0011	20	TBIT	JSR	DELAY
0012	30			
0013	00			
0014	38		SEC	
0015	6A		ROR	A
0016	2E		ROL	VIAORB
0017 0018	VIAORB			
0019	CA		DEX	
001A	D0		BNE	TBIT
001B	F5			
001C	00		BRK	

In tatsächlichen Anwendungen werden Sie finden, daß es erforderlich ist, eine Eins auf die Fernschreibleitung nach jeder Konfiguration zu plazieren, da die Leitung im Eins-(Mark)-Zustand sein sollte, wenn keine Daten gesendet werden.

Jedes Zeichen besteht aus 11 Bits, beginnend mit einem Start-Bit (Null) und endend mit zwei Stop-Bits (Einsen).

Beachten Sie, daß Sie eine Parität durch Zählen der Bits erzeugen können, wie in Kapitel 6 gezeigt wird. Das Programm lautet:

```
        LDY   #0       ;BITZÄHLUNG = NULL
        LDA   $60      ;HOLE DATEN
CHBIT   BPL   CHKZ     ;IST NÄCHSTES DATENBIT 1?
        INY            ;JA, ADDIERE 1 ZUR BITZÄHLUNG
CHKZ    ASL   A        ;PRÜFE NÄCHSTE BIT-POSITION
        BNE   CHBIT    ;BIS ALLE BITS NULLEN SIND
        BRK
```

Das Indexregister Y enthält die Anzahl der 1-Bits in den Daten. Das niedrigstwertige Bit des Indexregisters Y ist deshalb ein gerades Paritätsbit.

UART

Diese Verfahren sind zwar gebräuchlich, jedoch so komplex, um einen speziellen LSI-Baustein zu rechtfertigen, den UART, oder Universal Asynchronous Receiver/Transmitter (Universeller asynchroner Empfänger/Sender)[24]. Der UART wird das Empfangs-Verfahren durchführen und Daten in paralleler Form, sowie das "Data-Ready"-Signal liefern. Er wird auch Daten in paralleler Form aufnehmen, das Sende-Verfahren ausführen und ein "Peripheral Ready"-Signal liefern, wenn er weitere Daten verarbeiten kann. UARTs besitzen zahlreiche andere Eigenschaften, und zwar:

1) Die Fähigkeit, unterschiedliche Bitlängen zu verarbeiten (gewöhnlich 5 bis 8), wahlweise Parität und die Anzahl der Stop-Bits (gewöhnlich 1, 1-1/2, und 2).
2) Anzeigen für Rahmungsfehler, Paritätsfehler und "Überlauf-Fehler" (Fehler durch das Lesen eines Zeichens, bevor ein anderes empfangen wird).
3) Kombatibilität mit RS-232[25], d.h. ein Ausgangs-Signal "Sende-Anforderung" (RTS = Request to Send), das die Anwesenheit von Daten zum Kommunikationsgerät anzeigt und ein Eingangs-Signal "Sende-Bereitschaft" (CTS = Clear to Send), das als Antwort auf RTS die Bereitschaft des Kommunikationsgerätes anzeigt. Es können auch Vorkehrungen für andere RS-232-Signale vorhanden sein, wie "Received Signal Quality", "Data Set Ready" oder "Data Terminal Ready".
4) Tristate-Ausgänge und Steuer-Kompatibilität mit einem Mikroprozessor.
5) Takt-Optionen, die dem UART gestatten, die ankommenden Daten mehrere Male abzutasten, um falsche Start-Bits und andere Fehler festzustellen.
6) Unterbrechungs-Möglichkeiten und Steuerungen.

UARTs dienen als vier parallele Ports: Ein Eingangs-Daten-Port, ein Ausgangsdaten-Port, ein Eingangs-Statusport und ein Ausgangs-Steuerport. Die Status-Bits enthalten Fehleranzeigen, sowie Bereit-Flags (Ready-Flags). Die Steuer-Bits wählen verschiedene Optionen aus. UARTs sind preisgünstig (10.– bis 100.– DM, abhängig von ihren Eigenschaften) und leicht zu verwenden.

DER ASYNCHRONE KOMMUNIKATIONS-INTERFACE-ADAPTER 6850 (ACIA)[26,27]

ACIA-REGISTER

Der ACIA 6850 oder asynchrone Kommunikations-Interface-Adapter (siehe Bild 11-36) ist ein UART, der speziell für die Verwendung in Mikrocomputer-Systemen mit dem 6800 und 6502 entwickelt wurde. Er belegt zwei Speicher-Adressen und enthält zwei "Nur-Lese"-Register: (empfangene Daten und Status) und zwei "Nur-Schreib"-Register (gesendete Daten und Steuerung). Die Tabellen 11-16 und 11-17 beschreiben den Inhalt dieser Register.

SPEZIELLE EIGENSCHAFTEN DES ACIA

Beachten Sie die folgenden speziellen Eigenschaften des ACIA:

1) Lese- und Schreib-Zyklen adressieren physikalisch bestimmte Register. Deshalb kann man die ACIA-Register nicht als Adressen für Befehle wie "Inkrementiere", "Dekrementiere" oder "Verschiebe" verwenden, die sowohl Lese- wie Schreibzyklen enthalten.
2) Das ACIA-Steuerregister kann nicht von der CPU gelesen werden. Man muß eine Kopie des Steuerregisters im Speicher aufbewahren, wenn das Programm seinen Wert benötigt.
3) Der ACIA besitzt keinen RESET-Eingang. Er kann nur durch das Plazieren von Einsen in die Steuerregister-Bits 0 und 1 gelöscht werden. Dieses Verfahren (genannt MASTER RESET) ist erforderlich, bevor der ACIA eingesetzt wird, um zu vermeiden, daß er ein zufälliges Start-Zeichen besitzt.
4) Die RS-232-Signale sind alle aktiv Low. Request-to-Send (RTS) sollte insbesonders auf High gebracht werden, um es inaktiv zu machen, wenn es nicht in Verwendung steht.
5) Der ACIA benötigt einen externen Takt. Gewöhnlich werden 1760 Hz zugeführt und die Betriebsart ÷16 (Steuerregister-Bit 1 = 0, Bit 0 = 1) verwendet. Der ACIA wird den Takt zur Zentrierung des Empfangs verwenden, um falsche Start-Bits zu vermeiden, die durch Rauschen auf den Leitungen verursacht werden könnten.
6) Das "Data-Ready"-Flag (Empfangsdaten-Register voll, oder Receive-Data-Register-Full = RDRF) ist Bit 0 des Status-Registers. Das Peripheral-Ready-Flag (Sendedaten-Register leer, oder Transmit-Data-Register-Empty = TDRE) ist Bit 1 des Status-Registers.

Tabelle 11-16. Definition des Inhalts der ACIA-Register.

Daten-bus Zeilen-Nummer	Puffer-Adresse			
	RS · $\overline{R/W}$ Sende-Daten-Register	RS · R/W Empfangs-Daten-Register	$\overline{RS}$ · $\overline{R/W}$ Steuer-Register	$\overline{RS}$ · R/W Status-Register
	(Nur Schreiben)	(Nur Lesen)	(Nur Schreiben)	(Nur Lesen)
0	Data Bit 0*	Data Bit 0	Counter Divide Select 1 (CR0)	Receive Data Register Full (RDRF)
1	Data Bit 1	Data Bit 1	Counter Divide Select 2 (CR1)	Transmit Data Register Empty (TDRE)
2	Data Bit 2	Data Bit 2	Word Select 1 (CR2)	Data Carrier Detect ($\overline{DCD}$)
3	Data Bit 3	Data Bit 3	Word Select 2 (CR3)	Clear-to-Send ($\overline{CTS}$)
4	Data Bit 4	Data Bit 4	Word Select 3 (CR4)	Framing Error (FE)
5	Data Bit 5	Data Bit 5	Transmit Control 1 (CR5)	Receiver Overrun (OVRN)
6	Data Bit 6	Data Bit 6	Transmit Control 2 (CR6)	Parity Error (PE)
7	Data Bit 7***	Data Bit 7**	Receive Interrupt Enable (CR7)	Interrupt Request (IRQ)

* Führendes Bit = LSB = Bit 0
** Datenbit wird null in der Betriebsart 7 Bit plus Parität sein
*** Datenbit ist "beliebig" in der Betriebsart 7 Bit plus Parität

Tabelle 11-17. Bedeutung der ACIA-Steuerregister-Bits.

CR6	CR5	Funktion
0	0	$\overline{RTS}$ = Low, Sende-Unterbrechung gesperrt.
0	1	$\overline{RTS}$ = Low, Sende-Unterbrechung freigegeben.
1	0	$\overline{RTS}$ = High, Sende-Unterbrechung gesperrt.
1	1	$\overline{RTS}$ = Low. Sendet ein Signal für eine Unterbrechung auf dem Sende-Datenausgang. Sende-Unterbrechung gesperrt.

CR4	CR3	CR2	Funktion
0	0	0	7 Bits + Gerade Parität + 2 Stop-Bits
0	0	1	7 Bits + Ungerade Parität + 2 Stop-Bits
0	1	0	7 Bits + Gerade Parität + 1 Stop-Bit
0	1	1	7 Bits + Ungerade Parität + 1 Stop-Bit
1	0	0	8 Bits + 2 Stop-Bits
1	0	1	8 Bits + 1 Stop-Bit
1	1	0	8 Bits + Gerade Parität + 1 Stop-Bit
1	1	1	8 Bits + Ungerade Parität + 1 Stop-Bit

CR1	CR0	Funktion
0	0	÷ 1
0	1	÷ 16
1	0	÷ 64
1	1	Haupt-Reset

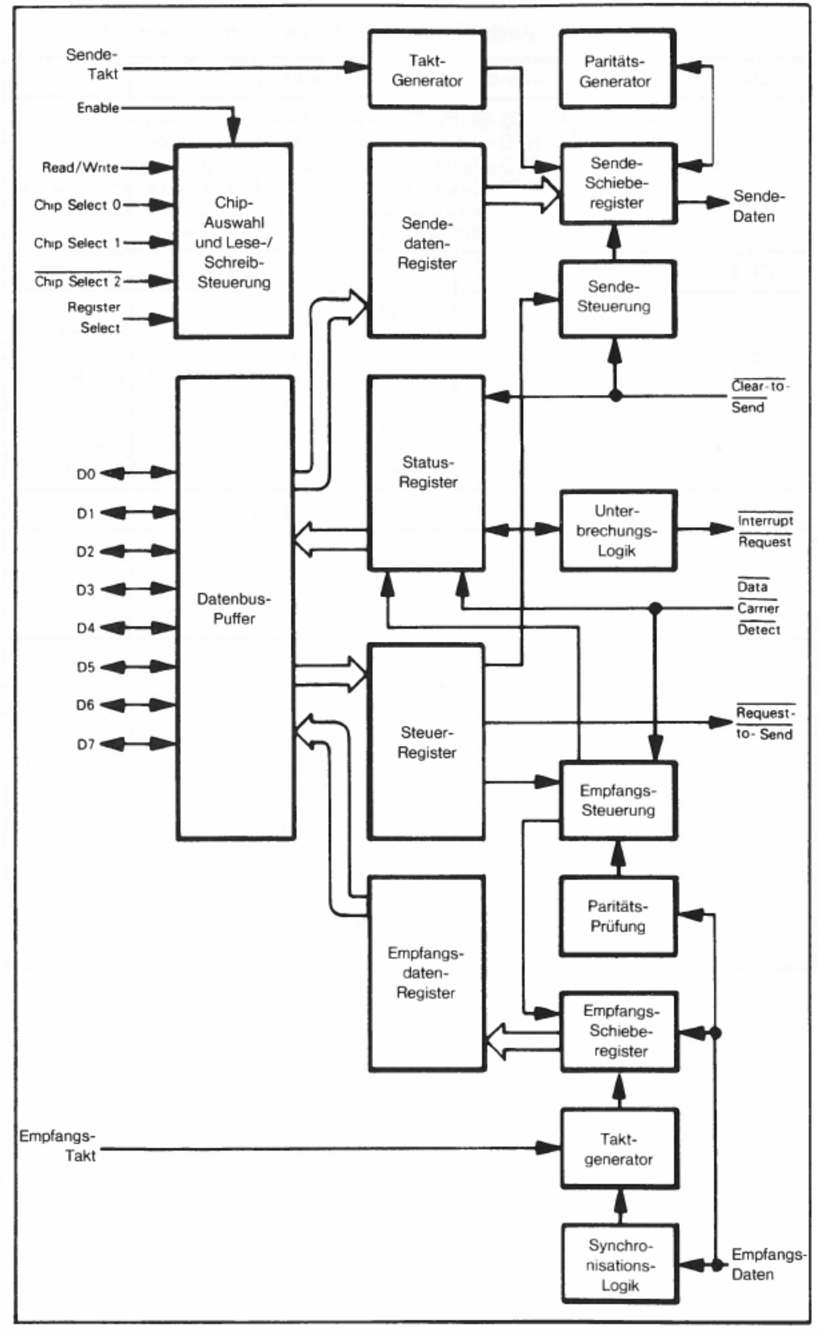


Bild 11-36. Blockschaltbild der ACIA 6850.

Aufgabe: Empfange Daten von einem Fernschreiber über einen ACIA 6850 und plaziere die Daten in den Speicherplatz 0040.

Quellprogramm:

```
        LDA   #%00000011  ;MASTER RESET FÜR ACIA
        STA   ACIACR
        LDA   #%01000101  ;KONFIGURIERE ACIA FÜR TTY MIT
                          ;  UNGERADER PARITÄT
        STA   ACIACR
WAITD   LDA   ACIASR      ;HOLE ACIA-STATUS
        LSR   A           ;WURDEN DATEN EMPFANGEN?
        BCC   WAITD       ;NEIN, WARTE
        LDA   ACIADR      ;JA, HOLE DATEN VOM ACIA
        STA   $60         ;BEWAHRE DATEN AUF
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A9		LDA	#%00000011
0001	03			
0002	8D		STA	ACIACR
0003 0004	ACIACR			
0005	A9		LDA	#%01000101
0006	45			
0007	8D		STA	ACIACR
0008 0009	ACIACR			
000A	AD	WAITD	LDA	ACIASR
000B 000C	ACIASR			
000D	4A		LSR	A
000E	90		BCC	WAITD
000F	FA			
0010	AD		LDA	ACIADR
0011 0012	ACIADR			
0013	85		STA	$60
0014	60			
0015	00		BRK	

Das Programm muß den ACIA anfangs löschen, indem es Einsen in die Steuerregister-Bits 0 und 1 plaziert. Der ACIA besitzt kein internes Löschen beim Einschalten der Betriebsspannung, das den ACIA im Reset-Status hält, bis der "Master-Reset" zugeführt wird.

BEISPIEL FÜR ACIA-KONFIGURATIONEN

Dieses Programm konfiguriert das ACIA-Steuer-Register wie folgt:

Bit 7 = 0, um die Empfänger-Unterbrechung zu sperren

Bit 6 = 1, um Request-to-Send (RTS) High (inaktiv) zu machen.

Bit 5 = 0, um die Sender-Unterbrechung zu sperren.

Bit 4 = 0, für 7-Bit-Worte

Bit 3 = 0, Bit 2 = 1 für ungerade Parität mit zwei Stop-Bits

Bit 1 = 0, Bit 0 = 1 für Takt dividiert durch 16 (es müssen 1760 Hz zugeführt werden.)

Das empfangene Daten-Statusbit ist Statusregister-Bit 0. Was würde geschehen, wenn wir versuchen

```
LDA   ACIASR
LSR   A
```

durch

```
LSR   ACISR
```

zu ersetzen?

Erinnern wir uns daran, daß sich die Status- und Steuerregister eine Adresse teilen, jedoch physikalisch voneinander getrennt sind.

Versuchen Sie eine Fehlerprüf-Routine in das Programm einzufügen. Setze:

(0061) = 0, wenn keine Fehler auftraten
= 1, wenn ein Paritätsfehler auftrat (Statusregister-Bit 6 = 1)
= 2, wenn ein Überlauf-Fehler auftrat (Statusregister-Bit 5 = 1)
= 3, wenn ein Rahmungsfehler auftrat (Statusregister-Bit 4 = 1)

Nehmen Sie an, daß die Priorität von Fehlern vom MSB zum LSB im ACIA-Statusregister geht (d.h. Paritätsfehler haben die Priorität vor Überlauffehlern, die wiederum Priorität vor Rahmungsfehlern haben, wenn mehr als ein Fehler aufgetreten ist).

Aufgabe: **Sende Daten vom Speicherplatz 0040 zu einem Fernschreiber über einen ACIA 6850.**

Quellprogramm:

```
        LDA   #%00000011 ;MASTER-RESET FÜR ACIA
        STA   ACIACR
        LDA   #%01000101 ;KONFIGURIERE ACIA FÜR TTY MIT
                         ;  UNGERADER PARITÄT
        STA   ACIACR
        LDA   #%00000010
WAITR   BIT   ACIASR     ;IST DER ACIA BEREIT FÜR DATEN?
        BEQ   WAITR      ;NEIN, WARTE BIS ES DER FALL IST
        LDA   $60        ;JA, HOLE DATEN
        STA   ACIADR     ;UND SENDE SIE
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
0000	A9		LDA	#%00000011
0001	03			
0002	8D		STA	ACIACR
0003 0004	ACIACR			
0005	A9		LDA	#%01000101
0006	45			
0007	8D	STA	ACIACR	
0008 0009	ACIACR			
000A	A9		LDA	#%00000010
000B	02			
000C	2C	WAITR	BIT	ACIASR
000D 000E	ACIASR			
000F	F0		BEQ	WAITR
0010	FB			
0011	A5		LDA	$60
0012	60			
0013	8D		STA	ACIADR
0014 0015	ACIADR			
0016	00		BRK	

Das Sender-Statusbit ist das Statusregister-Bit 1. Wie könnten Sie das Empfangsprogramm modifizieren, um den Bit-Test-Befehl zu verwenden?

DER ASYNCHRONE KOMMUNIKATIONS-INTERFACE-ADAPTER (ACIA) 6551

6551 ACIA-REGISTER

Der 6551 ACIA ist eine Variante des Bausteins 6850, der sowohl in Systemen mit dem 6800 wie dem 6502 verwendet werden kann. Bild 11-37 zeigt ein Blockschaltbild dieses Bausteins. **Er besitzt die meisten Eigenschaften des ACIA 6850 und enthält auch einen Baud-Raten-Generator, der 15 programmierbare Baudraten, abgeleitet von einem externen Quarz mit 1.8432 MHz liefern kann. Daher kann der ACIA 6551 nahezu jede gebräuchliche Baudrate ohne externen Zeitgeber oder Baudraten-Generator liefern.** Dieser Baustein besitzt vier interne Register, die gemäß Tabelle 11-18 adressiert werden. Seine Arbeitsweise wird durch zwei Register gesteuert.

1) Das Steuerregister (siehe Bild 11-38) steuert den Baudraten-Generator, die Wortlänge, die Anzahl der Stopbits, und die Taktquelle des Empfängers.

BEISPIEL EINER KONFIGURATION DES ACIA 6551

2) Das Kommandoregister (siehe Bild 11-39) steuert die Paritäts-Prüfung und Erzeugung, Unterbrechungs-Freigabe und die Quittierungssignale für RS-232. Beachten Sie, daß das Programm den ACIA 6551 jederzeit zurücksetzen kann, indem irgendwelche Daten in die Adresse des Statusregisters (siehe Bild 11-40) geschrieben werden. Zum Beispiel löscht das folgende Programm den ACIA 6551 und konfiguriert ihn für einen Fernschreiber mit 10 Zeichen pro Sekunde, ungerader Parität und zwei Stop-Bits:

```
LDA   #%10110011
STA   ACIASR        ;RESET FÜR 6551 ACIA
STA   ACIAMR        ;KONFIGURIERE BETRIEBSART FÜR TTY
                    ;  (7 BITS, 2 STOPBITS)
LDA   #%00100011
STA   ACIACR        ;KONFIGURIERE UNGERADE PARITÄT,
                    ;  KEINE UNTERBRECHUNGEN
```

Wir haben dem Steuer- (Betriebsart)-Register den Namen ACIAMR gegeben.

Das Programm konfiguriert das Steuer (Betriebsart)-Register des ACIA 6551 wie folgt:

Bit 7 = 1, für zwei Stopbits
Bit 6 = 0, Bit 5 = 1 für 7-Bit-Worte
Bit 4 = 1, um einen Empfänger-Takt vom internen Generator zu erzeugen
Bits 0 – 3 = 0011 für 109.92 Baud (10 Zeichen pro Sekunde) vom internen Baudraten-Generator

Das Programm konfiguriert das Kommandoregister des ACIA 6551 wie folgt:

Bit 7 = 0, Bit 6 = 0, Bit 5 = 1 für ungerade Parität sowohl für Empfänger wie Sender
Bit 4 = 0, damit Zeichen nicht automatisch zurück durch den Sender ausgesandt werden
Bit 3 = 0, Bit 2 = 0, um die Sender-Unterbrechung zu sperren und $\overline{RTS}$ auf High (inaktiv) zu bringen
Bit 1 = 1, um die Empfänger-Unterbrechung zu sperren (dies ist ein Maskenbit)
Bit 0 = 1, um den Empfänger/Sender freizugeben

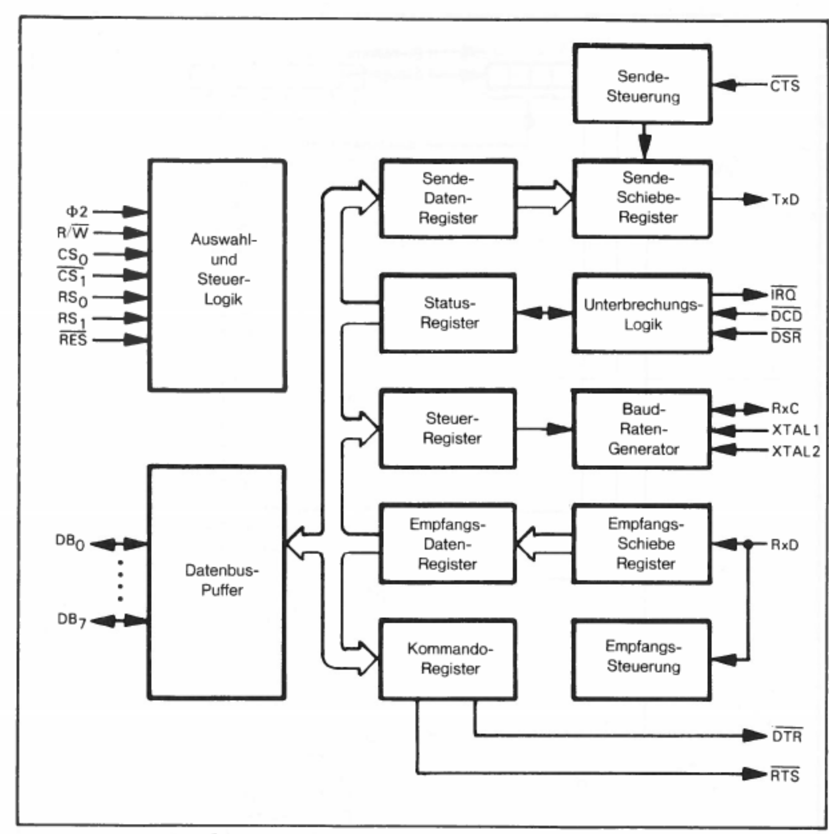


Bild 11-37. Blockschaltbild des ACIA 6551.

Tabelle 11-18. Adressierung der internen Register des ACIA 6551.

RS_1	RS_0	Schreiben	Lesen
0	0	Sende-Daten-Register	Empfangs-Daten-Register
0	1	Programmiertes Löschen (Daten sind "beliebig")	Status-Register
1	0	Kommando-Register	
1	0	Steuer-Register	

Die Tabelle zeigt, daß nur in das Kommando- und Steuer-Register geschrieben oder aus diesen gelesen werden kann. Die Operation "Programmiertes Löschen" bewirkt keinerlei Daten-Transfer, wird jedoch zum Löschen der Register des SY6551 verwendet. Das programmierte Löschen unterscheidet sich etwas vom Hardware-Reset ($\overline{RES}$), und diese Unterschiede werden bei den Definitionen der individuellen Register beschrieben.

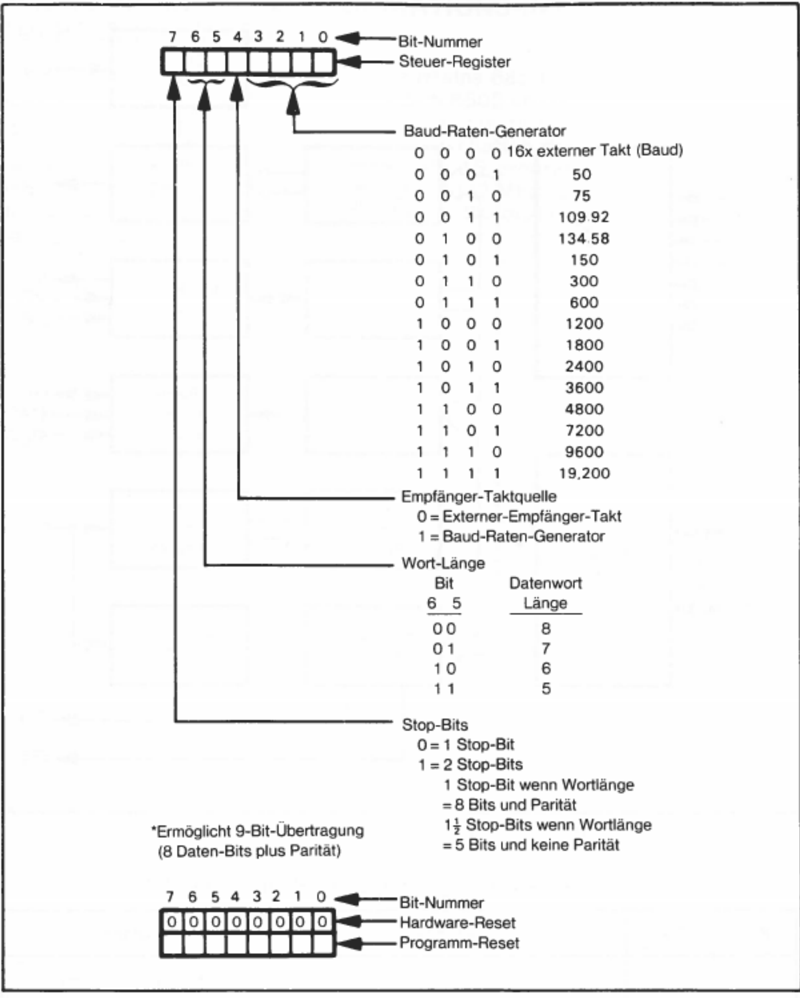


Bild 11-38. Definition des Inhalts des Steuer-Registers des ACIA 6551.

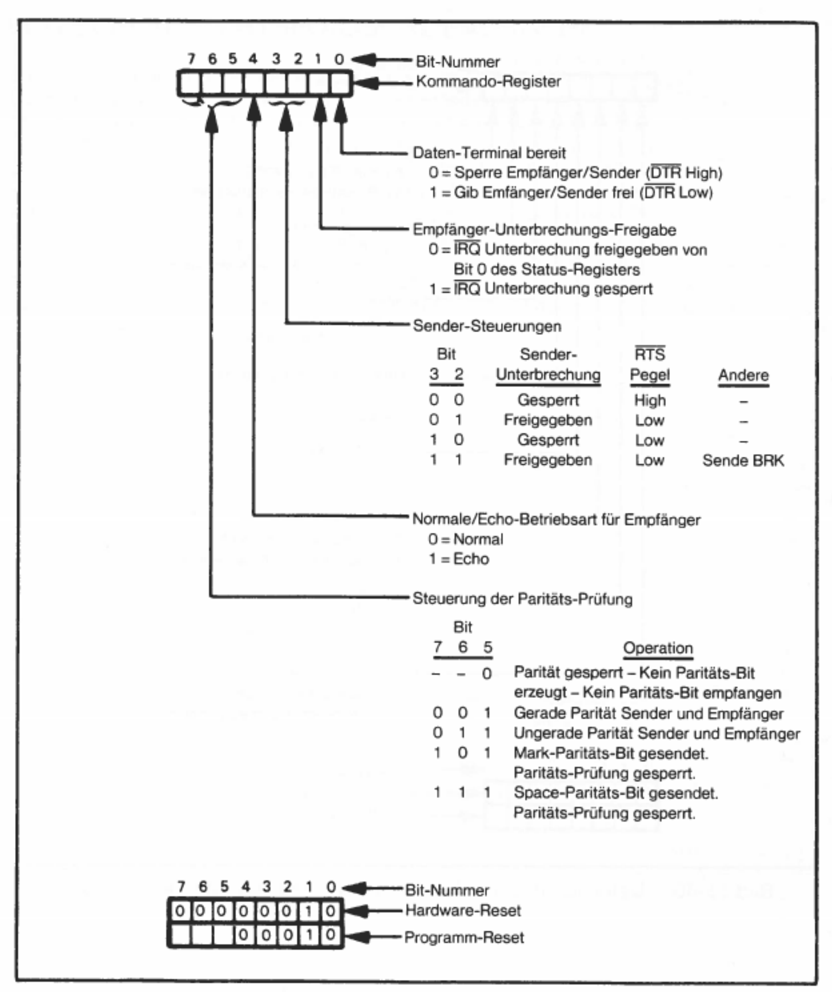


Bild 11-39. Definition des Inhalts des Kommando-Registers des ACIA 6551.

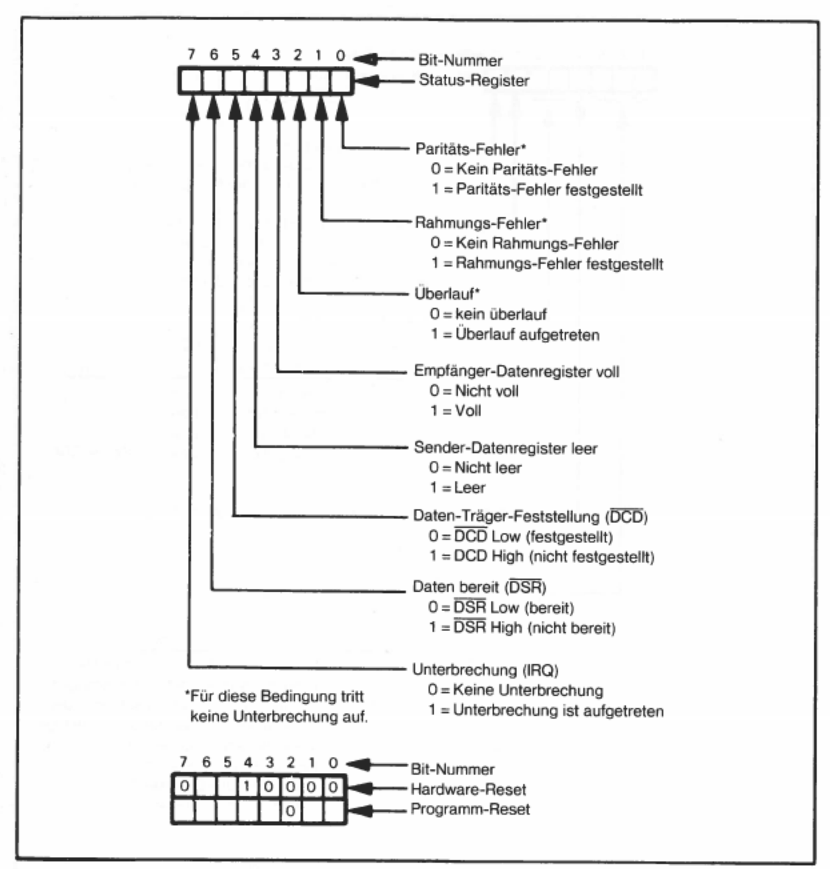


Bild 11-40. Definition des Inhalts des Status-Registers des ACIA 6551.

LOGISCHE UND PHYSIKALISCHE BAUSTEINE[28]

LOGISCHE BAUSTEINE

Ein wesentlicher Vorteil beim Schreiben von E/A-Routinen besteht darin, sie unabhängig von spezieller physikalischer Hardware zu machen. Die Routinen können dann Daten zu oder von E/A-Bausteinen transferieren, wobei die tatsächlichen Adressen als Parameter geliefert werden. Auf den E/A-Baustein kann in Wirklichkeit über ein spezielles Interface zugegriffen werden, der als physikalischer Baustein bezeichnet wird. Der E/A-Baustein, zu dem das Programm Daten transferiert, wird als logischer Baustein bezeichnet. Das Betriebssystem oder Überwachungsprogramm muß eine Anordnung (mapping) der logischen Bausteine liefern, die auf den physikalischen Bausteinen liegen, das heißt, tatsächliche physikalische E/A-Adressen und Eigenschaften so zuzuordnen, daß sie von E/A-Routinen verwendet werden können.

Beachten Sie die Vorteile dieser Lösung:

1) Das Betriebssystem kann die Zuweisung unter der Steuerung des Anwenders variieren. Beachten Sie, daß der Anwender leicht einen Test-Panel oder ein Entwicklungssystem-Interface gegen die tatsächlichen E/A-Bausteine austauschen kann. Dies ist sehr nützlich bei Wartungsarbeiten, sowie bei der Fehlersuche und beim Testen. Ferner kann der Anwender die E/A-Geräte für verschiedene Situationen austauschen. Typische Beispiele wären die Ausgabe von Zwischenergebnissen zu einer Video-Anzeige und die endgültige Ausgabe zu einem Drucker oder die Eingabe von einer Fernsteuerungs-Leitung anstatt von einer lokalen Tastatur.

2) Die gleichen E/A-Routinen können mehrere identische oder ähnliche Bausteine handhaben. Das Betriebssystem oder der Anwender braucht zum Beispiel nur die Adresse eines speziellen Fernschreibers, RS-232 Terminals oder Druckers zu liefern.

3) Änderungen, Korrekturen oder Ergänzungen zur E/A-Konfiguration sind leicht auszuführen, da nur die Zuweisungen (oder die Anordnungen) geändert werden müssen.

Beim Mikroprozessor 6502 kann entweder die vor-indizierte (indizierte indirekte) oder nach-indizierte (indirekte indizierte) Adressier-Art bei den E/A-Routinen verwendet werden, um eine Unabhängigkeit von speziellen physikalischen Adressen zu erreichen. Vor-Indizierung ist bequem, da sie die Auswahl verschiedener physikalischer Baustein-Adressen von einer Tabelle gestatten.

E/A-BAUSTEIN-TABELLE

Wenn eine Tabelle von E/A-Adressen auf Seite Null aufbewahrt wird, ist alles, was eine E/A-Adresse benötigt, ein Index in diese Tabelle. Sie kann dann auf den E/A-Baustein unter Verwendung der vor-indizierten (oder indizierten indirekten) Adressier-Art zugreifen. Wenn zum Beispiel die Baustein-Nummer im Speicherplatz DEV liegt, würde das Programm zur Berechnung des Index lauten:

```
        LDA   DEV          ;HOLE BAUSTEIN NUMMER
        ASL   A            ;MULTIPLIZIERE MIT 2 FÜR 2-BYTE-
                           ; ADRESSENTABELLEN
        TAX
```

Daten können nun von oder zum entsprechenden E/A-Baustein mit folgenden Befehlen transferiert werden

```
        LDA   DATA         ;HOLE DATEN
        STA   (IOTBL,X)    ;SENDE DATEN ZUM LOGISCHEN
                           ; E/A-BAUSTEIN
```

oder

```
        LDA   (IOTBL,X)    ;HOLE DATEN VOM LOGISCHEN
                           ; E/A-BAUSTEIN
        STA   DATA         ;BEWAHRE DATEN AUF
```

Die gleiche E/A-Routine kann Daten zu oder von mehreren verschiedenen E/A-Bausteinen transferieren, indem sie einfach mit unterschiedlichen Indizes versehen werden. Vergleichen Sie die Flexibilität dieser Lösung mit der geringen Flexibilität von E/A-Routinen, die direkte Adressierung verwenden und deshalb zu speziellen physikalischen Adressen geführt werden.

STANDARD-INTERFACES

STANDARD-INTERFACES

Andere Standard-Interfaces neben der TTY-Stromschleife und RS-232 können ebenfalls verwendet werden, um periphere Geräte mit dem Mikrocomputer zu verbinden. Einige bekannte sind:

1) Die seriellen Interfaces RS-449, RS-422 und RS-423.[29]

2) Der parallele universelle Interface-Bus (General-Purpose-Interface-Bus = GPIB) auch bekannt als IEEE-488 oder Hewlett-Packard Interface-Bus (HPIB).[30]

3) Der Hobby-Computerbus oder Altair/Imsai-Bus S-100.[31] Dies ist ebenfalls ein 8-Bit-Bus.

4) Der Intel-Multibus.[32] Dies ist ein weiterer 8-Bit-Bus, der jedoch erweitert werden kann, so daß er 16 Bits parallel verarbeitet.

AUFGABEN:

1) Trennung von Tastenbetätigungen von einer nicht codierten Tastatur

Zweck: Das Programm sollte Eingaben von einer nicht codierten 3 x 3-Tastatur lesen und sie in eine Anordnung plazieren. Die Anzahl der erforderlichen Eingaben befinden sich im Speicherplatz 0040 und die Anordnung beginnt im Speicherplatz 0041.

Trennen Sie eine Tastenbetätigung von der nächsten, indem Sie auf das Ende der momentanen Tastenbetätigung warten. Vergessen Sie nicht, die Tastatur zu entprellen (dies kann einfach ein Warten von 1 ms sein).

Beispiel:

(0040) = 04

Eingaben sind 7, 2, 2, 4

Ergebnis:

(0041) = 07
(0042) = 02
(0043) = 02
(0044) = 04

2) Lesen eines Satzes von einer codierten Tastatur

Zweck: Das Programm sollte die Eingaben von einer ASCII-Tastatur lesen (7 Bits mit einem Null-Paritätsbit) und es in eine Anordnung plazieren, bis es einen ASCII-Punkt $2E_{16}$ empfängt. Die Anordnung beginnt im Speicherplatz 0040. Jede Eingabe ist durch einen Tast-Impuls markiert, wie in dem Beispiel "Eine codierte Tastatur".

Beispiel:

Eingaben sind H, E, L, L, O,.

Ergebnis:

(0040) = 48 H
(0041) = 45 E
(0042) = 4C L
(0043) = 4C L
(0044) = 4F O
(0045) = 2E .

3) Ein Rechteck-Generator mit variabler Amplitude

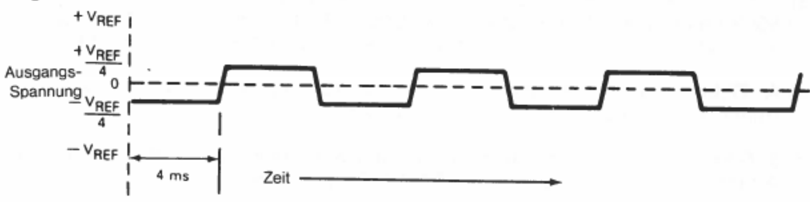
Zweck: Das Programm sollte eine Rechteck-Welle erzeugen, wie im nächsten Bild gezeigt ist, unter Verwendung eines D/A-Wandlers. Der Speicherplatz 0040 enthält die geeichte Amplitude der Welle, der Speicherplatz 0041 die Länge eines Halbzyklus in Millisekunden, und der Speicherplatz 0042 die Anzahl der Zyklen.

Nehmen Sie an, daß eine digitale Ausgabe von 80_{16} zum Konverter in einem analogen Ausgangssignal von null Volt resultiert. Im allgemeinen ergibt eine digitale Ausgabe von D ein analoges Ausgangssignal von (D-80)/80 X $-V_{REF}$ Volt.

Beispiel:

(0040) = A0 (hex)
(0041) = 04
(0042) = 03

Ergebnis:



Die Basis-Spannung ist 80_{16} = 0 Volt. Die volle Skala beträgt $100_{16} = -V_{REF}$ Volt.
Daher ist $A0_{16}$ = (A0–80)/80 X $-V_{REF} = -V_{REF}/4$.

Das Programm erzeugt 3 Impulse mit der Amplitude $V_{REF}/4$ mit einer halben Zyklus-Länge von 4 ms.

4) Mittelwert-Bildung von analogen Ablesungen

Zweck: Das Programm sollte vier Ablesungen von einem A/D-Wandler im Abstand von 10 Millisekunden entnehmen und den Mittelwert in den Speicherplatz 0040 legen. Es werde angenommen, daß der A/D-Wandler 100 Mikrosekunden für eine Umwandlung benötigt, so daß die Umwandlungszeit ignoriert werden kann.

Beispiel:

Ablesungen sind (hex) 86, 89, 81, 84

Ergebnis: (0040) = 85

5) Ein Terminal mit 30 Zeichen pro Sekunde

Zweck: Modifiziere die Sende- und Empfangs-Routinen für das Beispiel "Ein Fernschreiber", um ein Terminal mit 30 Zeichen/Sekunde zu steuern, das ASCII-Daten mit einem Stop-Bit und gerader Parität überträgt. Wie könnten Sie die Routinen schreiben, um jedes Terminal zu steuern, abhängig von einem Flag-Bit im Speicherplatz 0060. D.h. Zeichen/Sekunde (0060) = 0 für das Terminal mit 30 Zeichen/Sekunde, (0060) = 1 für das Terminal mit 10 Zeichen/Sekunde?

LITERATUR

1) J. Barnes, and V. Gregory, "Use Microprocessors to Enhance Performance with Noisy Data," EDN August 20, 1976, pp. 71-72

2) J. E. McNamara, Technical Aspects of Data Communications (Maynard, Mass.: Digital Equipment Corporation, 1977), Chapter 13.

R. Swanson, "Understanding Cyclic Redundancy Codes," Computer Design, November 1975, pp. 93-99.

J. Wong, et al., "Software Error Checking Procedures for Data Communications Protocols," Computer Design, February 1979, pp. 122-125.

Der letzte Artikel enthält einige 6800-Assemblersprachen-Programme für CRC-Erzeugung.

3) For exaple, the 6844 Direct Memory Access Controller for 6800- or 6502-based microcomputers is decribed in An Introduction to Microcomputers: Volume 2 – Some Real Microprocessors, pp. 9-106 through 9-123.

4) A. Osborne, et al., An Introduction to Microcomputers: Volume 2 – Some Real Microprocessors, pp. 9-45 through 9-54.

J. Gilmore, and R. Huntington, "Designing with the 6802 Peripheral Interface Adapter," Electronics, December 23, 1976, pp. 85-86.

5) L. Leventhal, 6800 Assembly Language Programming, pp. 11-31 through 11-47, 11-49 through 11-74.

6) A. Osborne, et al., An Introduction to Mircrcomputers: Volume 2 – Some Real Microprocessors, pp. 10-29 through 1047.

7) R6500 Microcomputer System Hardware Manual (Anaheim, Calif.: Rockwell International), pp. 1-65 through 1-97.

8) W. C. Mavity, "Megabit Bubble Modules in on Mass Strogage," Electronics, March 29, 1979, pp. 99-103.

9) J. Gieryic, "SYM-1 6522-Based Timer," Micro, April 1979, pp. 11:31 through 11:32. The magazine Micro is dedicadet exclusively to 6502-based personal computers; it is available (monthly publication) from the COMPUTERIST, Inc., P. O. Box 3, South Chelmsford, MA 01824.

10) M. L. DeJong, "A Simple 24-Hour Clock for the AIM 65," Micro, March 1979, pp. 10:5 through 10:7.

11) A. Osborne, et al., An Introduction to Microcomputers: Volume 2 – Some Real Microprocessors, pp. 10-47 through 10-55.

12) C. Forster, Programming a Microcomputer: 6502 (Reading, Mass.: Addison-Wesley, 1978). This is a very elementary introduction to computers based on the KIM microcomputer.

13) R. C. Camp, et al., Microcomputer Systems Principles Featuring the 6502/KIM (Portland: Matrix Publishers, 1978).

14) A. Caprihan, et al., "A Simple Microcomputer for Biomedical Signal Processing," 4th Annual Conference on Industrial Applications of Microprocessors, 1978, pp. 18-23. Proceedings (since 1975) are available from IEEE, 445 Hoes Lane, Piscataway, NJ 08854.

15) The TTL Data Book for Design Engineers, Texas Instruments Inc., P. O. Box 5012, Dallas, TX 75222, 1976, pp. 7-151 through 7-156.

16) E. Dilatush, "Special Report: Numeric and Alphanumeric Displays," EDN, February 5, 1978, pp. 26-35.

17) See Reference 15, pp. 7-22 through 7-34.

18) M. L. DeJong, "6502 Interfacing for Beginners: an ASCII Keyboard Input Port," Micro, February 1979, pp. 9-11 through 9-13.

19) E. R.Hnatek, A User's Handbook of D/A and A/D-Converters (New York: Wiley 1976).

20) J. Kane, et al., An Introduction to Microcomputers: Volume 3 – Some Real Support Devices, Section E.

21) M. L. DeJong, "Digital-Analog and Analog-Digital Conversion Using the KIM-1," The Best of Micro, Volume 1, pp. 30-33.

22) P. H. Garrett, Analog Systems for Microprocessors and Minicomputers (Reston, VA.: Reston Publishing CO., 1978).

23) G. L. Zick and T. T. Sheffer, "Remote Failure Analysis of Micro-based Instrumentation," Computer, September 1977, pp. 30-35.

24) For a discussion of UARTs, see P. Rony et al., "The Bugbook IIa," E and L Instruments Inc., 61 First Street, Derby CT. 06418 or D. G. Larsen et al., "INWAS: Interfacingn with Asychronous Serial Mode," IEEE Transactions on Industrial Electronics and Control Instrumentation, February 1977, pp. 2-12. See also McNamara, Reference 2.

25) The official RS-232 standard is available as "Interface Between Data Terminal Equipment and Data Communications Equipment Employing Serial Binary Data Interchange" E1A RS-232C August 1969. You can find introductory descriptions of RS-232 in G. ickles, "Who's Afraid of RS-232?", Kilobaud, May 1977, pp. 50-4 and in C. A. Ogdin, "Microcomputer Buses – Part II," Mini-Micro Systems, July 1978, pp. 76-80. Ogdin also describes the new RS-449 standard.

26) A. Osborne, et al., An Introduction to Microcomputers: Volume 2 - Some Real Microprocessors, pp. 9-55 through 9-61.

27) K. Fronheiser, "Device Operation and System Implementation of the Asynchronous Communications Interface Adapter," Motorola Semiconductor Products Application Note AN-754, 1975.

28) C. W. Gear, Computer Organization an Programming 2/E (New York: McGraw-Hill, 1974), Chapter 6.

29) D. Morris, "Revised Data Interface Standards," Electronic Design, September 1, 1977, pp. 138-141.

30) Institute of Elctrial and Electronics Engineers, "IEEE Standard Digital Interface for Programmable Instrumentation," IEEE Std488-1978, IEEE, 445 Hoes Lane, Piscataway, NJ 08854.

J. B. Peatman, Microcomputer-Based Design, (New York: McGraw-Hill, 1977).

D. C. Loughry, and M. S. Allen, "IEEE Standard 488 and Microprocessor Synergism," Proceedings of the IEEE, February 1978, pp. 162-172.

31) G. Morrow, and H. Fullmer, "Proposed Standard for the S-100 Bus," Computer, May 1978, pp. 84-89.

M. L. Smith, "Build Your Own Interface," Kilobaud, June 1977, pp. 22-28

32) T. Rolander, "Intel Multibus Interfacing," Intel Application Note AP-28, Intel Corporation, Santa Clara, CA, 1977. See also An Introduction to Microcomputers: Volume 3 – Some Real support Devices, Section J.

Kapitel 12
UNTERBRECHUNGEN

Unterbrechungen sind Eingaben, die die CPU als Teil jedes Befehls-Zyklus prüft. Diese Eingaben gestatten der CPU, auf Vorgänge mehr auf Hardware-Ebene zu reagieren als auf Software-Ebene über das Prüfen jedes Bausteins (polling = Abfragen). Unterbrechungen benötigen im allgemeinen mehr Hardware als eine gewöhnliche (programmierte) E/A, ergeben jedoch eine schnellere und direktere Reaktion.[1]

BEGRÜNDUNG FÜR UNTERBRECHUNGEN

Weshalb verwendet man Unterbrechungen? **Unterbrechungen gestatten es, die unmittelbare Aufmerksamkeit der CPU auf Vorgänge zu erlangen,** wie etwa Alarm, Betriebsspannungs-Ausfall, das Einfügen von Zeit-Intervallen und periphere Bausteine, die über Daten verfügen oder bereit sind, Daten anzunehmen. **Der Programmierer muß nicht erst dafür sorgen, daß jede mögliche Datenquelle abgefragt wird, und braucht sich nicht um nicht vorhandene Vorgänge zu kümmern.** Ein Unterbrechungs-System entspricht etwa einer Telefonklingel, sie läutet, wenn ein Aufruf empfangen wird, so daß man den Hörer nicht laufend abnehmen muß um festzustellen, ob sich jemand in der Leitung befindet. Die CPU kann ihre normale Tätigkeit ausführen (und verschwendet dadurch weniger Zeit). Wenn irgend etwas geschieht, so macht sich die Unterbrechung bei der CPU bemerkbar und zwingt sie dazu, die Eingabe zu bedienen, bevor sie ihre normalen Operationen fortsetzt. Natürlich wird diese einfache Beschreibung wesentlich komplizierter (wie etwa das Schaltbrett eines Telefons), wenn es mehrere Unterbrechungen unterschiedlicher Wichtigkeit gibt, sowie Aufgaben, die nicht unterbrochen werden dürfen.

EIGENSCHAFTEN VON UNTERBRECHUNGS-SYSTEMEN

Die Ausführungen von Unterbrechungs-Systemen unterscheiden sich sehr stark. Unter den zu beantwortenden Fragen für die Charakterisierung eines speziellen Systems sind:

1) Wieviele Unterbrechungs-Eingänge gibt es?
2) Wie reagiert die CPU auf eine Unterbrechung?
3) Wie bestimmt die CPU die Quelle einer Unterbrechung, wenn die Anzahl der Quellen die Anzahl der Eingänge überschreitet?
4) Kann die CPU zwischen wichtigen und unwichtigen Unterbrechungen unterscheiden?
5) Wie und wann wird das Unterbrechungs-System freigegeben und gesperrt?

Es gibt zahlreiche unterschiedliche Antworten auf diese Fragen. **Der Zweck aller dieser Ausführungen besteht jedoch darin, daß die CPU rasch auf Unterbrechungen reagiert und danach wieder ihre normale Tätigkeit aufnimmt.**

Die Anzahl der Unterbrechungs-Eingänge auf dem CPU-Chip bestimmt die Anzahl der verschiedenen Reaktionen, die die CPU ohne zusätzliche Hardware oder Software hervorbringen kann. Jeder Eingang kann eine unterschiedliche interne Reaktion bewirken. Unglücklicherweise besitzen die meisten Mikroprozessoren eine sehr kleine Anzahl (typisch 1 oder 2) getrennter Unterbrechungs-Eingänge.

Die eigentliche Reaktion der CPU auf eine Unterbrechung muß die Übergabe der Steuerung zur richtigen Unterbrechungs-Service-Routine sein und die Aufbewahrung des momentanen Wertes des Befehlszählers. Die CPU muß daher eine Befehl "Springe zu Unterprogramm" oder Aufrufbefehl mit dem Beginn der Unterbrechungs-Service-Routine an dieser Adresse ausführen. Diese Aktion wird die Rückkehr-Adresse im Stapel aufbewahren und die Steuerung der Unterbrechungs-Service-Routine übergeben. Der Umfang der benötigten externen Hardware für diese Reaktion variiert in hohem Maße. Einige CPUs erzeugen den Befehl und die Adresse intern. Andere benötigen externe Hardware, um dies auszuführen. Die CPU kann nur einen unterschiedlichen Befehl oder Adresse für jeden separaten Eingang erzeugen.

ABFRAGEN

VEKTORISIEREN

Wenn die Anzahl der unterbrechenden Bausteine die Anzahl der Eingänge überschreitet, benötigt die CPU zusätzliche Hardware oder Software, um die Quelle der Unterbrechung zu identifzieren. Im einfachsten Fall kann die Software eine Abfrage-Routine (Polling-Routine) sein, die den Status der Bausteine prüft, von denen eine Unterbrechung zu erwarten ist. Der einzige Vorteil eines derartigen Systems gegenüber dem normalen Abfragen besteht darin, daß die CPU weiß, daß wenigstens ein Baustein aktiv ist. Die andere Lösung besteht aus zusätzlicher Hardware, um eine eindeutige Daten-Eingabe (oder "Vektor") für jede Quelle zu liefern. Die beiden Alternativen können auch gemischt verwendet werden. Die Vektoren können Gruppen von Eingängen identifizieren, von denen die CPU einen speziellen hiervon durch Abfragen (polling) identifizieren kann.

PRIORITÄT

Ein Unterbrechungs-System, das zwischen wichtigen und unwichtigen Unterbrechungen unterscheiden kann, wird ein "Prioritäts-Unterbrechungssystem" genannt. Interne Hardware kann so viele Prioritäts-Ebenen liefern, wie es Eingänge gibt. Externe Hardware kann zusätzliche Ebenen durch die Verwendung eines Prioritätsregisters und Komparators zur Verfügung stellen. Die externe Hardware läßt nicht zu, daß die Unterbrechung die CPU erreicht, außer ihre Priorität ist höher als der Inhalt des Prioritätsregisters. Ein Prioritäts-Unterbrechungssystem kann einen speziellen Weg für die Verarbeitung von Unterbrechungen mit niedriger Priorität erfordern, die während längerer Zeitperioden ignoriert werden können.

FREIGABE UND SPERRUNG VON UNTERBRECHUNGEN

Die meisten Unterbrechungs-Systeme können freigegeben oder gesperrt werden. In der Tat sperren die meisten CPUs automatisch Unterbrechungen, wenn ein RESET ausgeführt wird (so daß der Programmierer das Unterbrechungs-System konfigurieren kann) **und bei der Annahme einer Unterbrechung** (so daß die Unterbrechung nicht durch ihre eigene Unterbrechungs-Routine unterbrochen wird). Der Programmierer kann auch den Wunsch haben, Unterbrechungen zu sperren, während er Daten vorbereitet oder verarbeitet, eine Zeit-Steuerschleife ausführt oder eine Mehr-Wort-Operation vollzieht.

NICHT-MASKIERBARE UNTERBRECHUNG

Eine Unterbrechung, die nicht gesperrt werden kann (manchmal auch eine "nicht-maskierbare Unterbrechung" genannt), kann sehr nützlich sein, bei einem Betriebsspannungs-Ausfall zu warnen, ein Vorgang, der offensichtlich Vorrang vor allen anderen Aktivitäten besitzen muß.

NACHTEILE VON UNTERBRECHUNGEN

Die Vorteile von Unterbrechungen sind offensichtlich, es gibt jedoch auch Nachteile. Diese sind:

1) Unterbrechungs-Systeme können einen beträchtlichen Teil zusätzlicher Hardware benötigen.
2) Unterbrechungen benötigen noch Datentransfers unter der Programmsteuerung durch die CPU. Sie besitzt keine Geschwindigkeits-Vorteile wie etwa beim DMA.
3) Unterbrechungen sind zufällige Eingaben, die möglicherweise schwierig zu testen und fehlerfrei zu machen sind. Fehler können sporadisch auftreten und daher sehr schwer zu finden sein.[2]
4) Unterbrechungen können einen beträchtlichen Aufwand erfordern, wenn zahlreiche Register-Inhalte aufbewahrt werden müssen und die Quelle durch Abfragen bestimmt werden muß.

6502-Unterbrechungs-System

Die interne Reaktion des 6502 auf eine Unterbrechung ist etwas komplex. Das Unterbrechungs-System besteht aus:

6502 UNTERBRECHUNGS-EINGÄNGE

1) **Ein maskierbarer Unterbrechungs-Eingang (IRQ) mit aktive Low und ein nicht-maskierbarer Unterbrechungs-Eingang (NMI) mit aktiv Low.**
2) **Ein Unterbrechungs-Sperr- (oder Masken-) Bit, das die maskierbare Unterbrechung sperrt.** Wenn das Unterbrechungs-Sperrbit 1 ist, sind keine maskierbare Unterbrechungen erlaubt.
   Das I-Bit wird in Bit 2 des Prozessor-Status- (oder P-) Registers gespeichert.

REAKTION DES 6502 AUF UNTERBRECHUNGEN

Der 6502 prüft den momentanen Status des Unterbrechungs-Systems am Ende jedes Befehls. Wenn eine Unterbrechung aktiv und freigegeben ist, so ist die Reaktion folgende:

1) **Die CPU bewahrt den Befehlszähler (höchstwertige Bits zuerst) und das Statusregister im Stapel auf.** Bild 12-1 zeigt die Reihenfolge, in der diese Register aufgewahrt werden. Beachten Sie, daß der Akkumulator und die Indexregister nicht automatisch gespeichert werden.
2) **Die CPU sperrt die maskierbare Unterbrechung (IRQ).** Das heißt, sie setzt Bit 2 (I) des Statusregisters.
3) **Die CPU holt eine Adresse von einem spezifizierten Paar von Speicheradressen und legt diese Adresse in den Befehlszähler.** Tabelle 12-1 enthält die Adressenpaare, die den verschiedenen Eingängen und dem Break-Befehl zugeordnet sind.

SPEZIELLE EIGENSCHAFTEN DES 6502-UNTERBRECHUNGS-SYSTEMS

Beachten Sie die folgenden speziellen Eigenschaften des Unterbrechungs-Systems des 6502:

1) Der 6502 bewahrt automatisch den Befehlszähler und das Statusregister im Stapel auf. Erinnern Sie sich daran, daß das Statusregister das Unterbrechungsfreigabe-Flag und das Break-Kommando-Flag beinhaltet.
2) Der 6502 liefert keine externen Signale zur Anzeige, daß er eine Unterbrechung angenommen hat mit Ausnahme der Adressen, die er auf den Adressenbus plaziert.
3) Der 6502 besitzt keine speziellen internen Vorkehrungen, um die Quelle einer Unterbrechung zu bestimmen, wenn es mehrere Quellen gibt, die zum gleichen Eingang geführt werden.

Der 6502 besitzt die folgenden speziellen Befehle, um sein Unterbrechungs-System zu handhaben:

1) **CLI (Clear Interrupt Disable Bit)** löscht Bit 2 des Statusregisters und gibt daher die maskierbare Unterbrechung frei.
2) **SEI (Set Interrupt Disable Bit)** setzt Bit 2 des Statusregisters und sperrt daher die maskierbare Unterbrechung.
3) **BRK (Force Break)** setzt das Break-Kommandoflag, bewahrt den Befehlszähler und das Statusregister im Stapel auf, sperrt die maskierbare Unterbrechung und plaziert den Inhalt der Adressen FFFE und FFFF in den Befehlszähler.

4) **RTI (Return from Interrupt) speichert das Statusregister und den Befehlszähler vom Stapel zurück.** Das Ergebnis ist, daß die alten Werte in den Befehlszähler und in das Statusregister zurückgelegt werden (einschließlich des Unterbrechungs-Bits). **RTI unterscheidet sich von RTS (Return from Subroutine) darin, daß RTI sowohl das Statusregister wie auch den Befehlszähler zurückspeichert und RTI nicht 1 zur Rückkehr-Adresse addiert, wie dies RTS ausführt** (siehe Kapitel 11 für eine Besprechung von RTS).

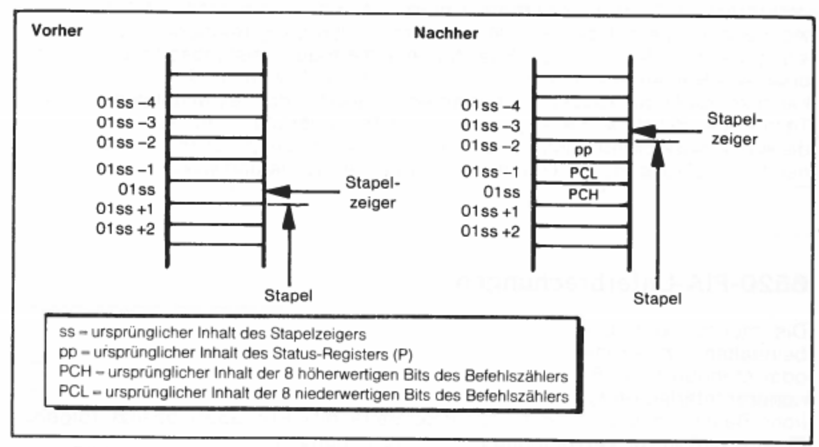


Bild 12-1. Aufbewahren des Status des Mikroprozessors im Stapel.

Tabelle 12-1. Speicherplan der 6502-Adressen, die zur Reaktion auf Unterbrechungen und Reset verwendet werden.

Quelle	Verwendete Adressen (Hexadezimal)
Unterbrechungs-Anforderung (IRQ) und BRK-Befehl	FFFE und FFFF
Reset (RESET)	FFFC und FFFD
Nicht-maskierbare Unterbrechung (NMI)	FFFA und FFFB

Die Adressen werden auf die gebräuchliche Art des 6502 gespeichert, mit den niedrigstwertigen Bits in der niedrigen Adresse.

BRK-BEFEHL

Der Befehl BRK (Force Break) erzeugt nahezu genau die gleiche Reaktion wie ein Unterbrechungs-Eingang (IRQ). Der einzige Unterschied besteht darin, daß das Break-Kommandoflag (Bit 4 des Statusregisters) gesetzt wird. Daher kann eine Service-Routine zwischen einem BRK-Befehl und einem IRQ-Eingang unterscheiden, indem sie Bit 4 des obersten Byte im Stapel prüft (erinnern Sie sich an Bild 12-1). Ein typisches Programm wäre:

```
PLA                   ;HOLE STATUSREGISTER VOM STAPEL
AND    #%00010000     ;IST BREAK-KOMMANDOFLAG GESETZT?
BNE    BREAK          ;JA, GEHE ZUR BREAK-ROUTINE
```

Der BRK-Befehl ist sehr nützlich bei der Fehlersuche (siehe Kapitel 14) und zur Rückgabe der Steuerung zu einem Monitor oder Betriebssystem. Siehe Kapitel 3 für weitere Informationen über den BRK-Befehl.

NICHT-MASKIERBARE UNTERBRECHUNG

Die nicht-maskierbare Unterbrechung ist ein flanken-sensitiver Eingang. Der Prozessor reagiert daher nur auf die Flanke eines Impulses auf dieser Leitung, und der Impuls wird seine eigene Unterbrechungs-Routine nicht unterbrechen. Nicht-maskierbare Unterbrechungen sind sehr nützlich bei Anwendungen, die auf den Ausfall der Stromversorgung reagieren müssen (d.h. sie müssen Daten in einen Speicher mit niedriger Leistungsaufnahme retten oder eine Reserve-Batterie zuschalten). Typische Anwendungen sind Kommunikationsgeräte, die Codes aufbewahren müssen und Teil-Mitteilungen, sowie Testgeräte, die teilweise abgeschlossene Tests verfolgen müssen. Wir werden die nicht-maskierbare Unterbrechung nicht weiter besprechen. Wir wollen annehmen, daß alle Unterbrechungs-Eingaben zu IRQ geführt wird.

6520-PIA-Unterbrechungen[3]

6520-PIA-UNTERBRECHUNGEN

Die meisten Unterbrechungssysteme des 6502 beinhalten programmierbare Interface-Chips oder Mehrfunktions-Bausteine, wie den 6520 Peripherer Interface Adapter, 6522 Versatile Interface-Adapter oder die Mehrfunktions-Bausteine 6530 und 6532. **Jede Seite des PIA 6520 besitzt folgende Eigenschaften zur Verwendung bei Unterbrechungen:**

1) **Einen Unterbrechungs-Eingang mit aktiv Low.**
2) **Unterbrechungs-Freigabe-Bits** (Bit 0 des Steuer-Registers für die Steuerleitung 1, Bit 3 für die Steuerleitung 2).
3) **Unterbrechungs-Status-Bits** (Bit 7 des Steuer-Registers für die Steuerleitung 1, Bit 6 für die Steuerleitung 2).

Die Bits 1 und 4 des Steuer-Registers bestimmen, ob eine ansteigende Flanke (Übergang Low auf High) oder eine Rückflanke (Übergang High auf Low) auf der Steuerleitung eine Unterbrechung verursacht.

Beachten Sie, daß:

1) **Die PIA-Unterbrechungs-Freigabebits die entgegengesetzte Polarität des I-Flags des 6502 besitzen,** d.h., sie müssen "1" sein, um eine Unterbrechung freizugeben.
2) **RESET löscht die PIA-Steuerregister und sperrt daher alle Unterbrechungen.**
3) **Die CPU kann die Bits 6 und 7 des Steuerregisters prüfen um festzustellen, ob ein PIA eine Unterbrechung bereit hat. Sobald gesetzt, bleiben diese Bits in diesem Zustand, bis die CPU die PIA-Datenregister liest.**
4) **Der PIA wird sich an eine Unterbrechung erinnern, die auftritt, während PIA-Unterbrechungen gesperrt waren, und wird eine Unterbrechungs-Anforderung liefern, sobald die PIA-Unterbrechung freigegeben wird.**

6522-VIA-UNTERBRECHUNGEN

6522-VIA-UNTERBRECHUNGEN

Der 6522 Versatile Interface-Adapter (Vielseitiger Interface-Adapter) kann auch als Unterbrechungsquelle verwendet werden. Dieser Baustein besitzt ein Unterbrechungs-Freigaberegister (Interrupt Enable Register = IER), das zur Freigabe der verschiedenen Unterbrechungsquellen verwendet werden kann, **und ein Unterbrechungsflag-Register (IFR = Interrupt Flag Register)**, das den Status der verschiedenen Quellen enthält. Bild 12-2 zeigt die Positionen der verschiedenen Freigabe-Bits im Unterbrechungs-Freigaberegister, und Bild 12-3 beschreibt das Unterbrechungsflag-Register.

FREIGABE UND SPERREN DER UNTERBRECHUNGEN DES 6522 VIA

Eine Unterbrechungsquelle kann freigegeben werden, indem die entsprechenden Freigabe-Bits gesetzt werden. Beachten Sie, daß das höchstwertige Bit kontrolliert, wie die anderen Freigabe-Bits beeinflußt werden.

1) Wenn IER7 = 0, löscht jede 1 in einer Bit-Position ein Freigabe-Bit und sperrt daher die Unterbrechung.
2) Wenn IER7 = 1, setzt jede 1 in einer Bit-Position ein Unterbrechungsbit und gibt daher diese Unterbrechung frei.

Nullen in den Freigabe-Bit-Positionen lassen die Freigabe-Bits unverändert.

Einige Beispiele für Freigabe und Sperren der 6522-VIA-Unterbrechungen sind:

1) Gib CA1-Unterbrechung frei, sperre alle übrigen:

```
LDA   #%01111101 ;SPERRE ALLE ÜBRIGEN
                 ; UNTERBRECHUNGEN
STA   VIAIER
LDA   #%10000010 ;GIB CA1-UNTERBRECHUNG FREI
STA   VIAIER
```

Die erste Operation setzt IER7 auf null, so daß die Einsen in den Bit-Positionen 0, 2, 3, 4, 5 und 6 die entsprechenden Freigabe-Bits löschen und daher diese Unterbrechungen sperren. Die zweite Operation setzt IER7 auf 1, so daß die 1 in Bit-Position 1 das entsprechende Freigabe-Bit (CA1-Unterbrechung) setzt und daher diese Unterbrechung freigibt.

2) Gib CB1- und CB2-Unterbrechungen frei, sperre alle übrigen.

```
LDA   #%01100111 ;SPERRE ALLE ÜBRIGEN
                 ; UNTERBRECHUNGEN
STA   VIAIER
LDA   #%10011000 ;GIB CB1-, CB2-UNTERBRECHUNGEN FREI
STA   VIAIER
```

Die erste Operation setzt IER7 auf 0, so daß die Einsen in den Bit-Positionen 0, 1, 2, 5 und 6 die entsprechenden Freigabe-Bits löschen und daher diese Unterbrechungen sperren. Die zweite Operation setzt IER7 auf 1, so daß die Einsen in den Bit-Positionen 3 und 4 die entsprechenden Freigabe-Bits (Bit 3 für CB2, Bit 4 für CB1) setzen und daher diese Unterbrechungen freigeben.

Neben den in Bild 12-3 beschriebenen Bedingungen können die Bits im Unterbrechungsflag-Register auch durch Schreiben von Einsen in die erforderlichen Bit-Positionen an der entsprechenden Adresse gelöscht werden. Dieses Verfahren ist sehr nützlich für das Löschen von Flags, die in den unabhängigen Betriebsarten verwendet werden, und zum Eliminieren unerwünschter Unterbrechungen, die zufällig während des Reset- oder Start-Vorganges verursacht werden könnten. Beachten Sie, daß die Bit-Positionen des Unterbrechungsflag-Registers die gleichen sind, wie die Bit-Positionen des Unterbrechungsfreigabe-Registers, so daß wir die vorhergehenden Beispiele leicht zum Eliminieren bestimmter Unterbrechungen erweitern können. Dies kann entweder durch Freigabe- oder Sperr-Operationen geschehen, da der Wert von Bit 7 keine Rolle spielt. Die erweiterten Beispiele sind:

1) Gib CA1-Unterbrechungen frei, sperre alle übrigen, lösche CA1-Unterbrechungs-Flag.

```
LDA   #%01111101  ;SPERRE ALLE ÜBRIGEN
                  ; UNTERBRECHUNGEN
STA   VIAIER
LDA   #%10000010
STA   VIAIFR      ;LÖSCHE CA1-UNTERBRECHUNGSFLAG
STA   VIAIER      ;GIB CA1-UNTERBRECHUNG FREI
```

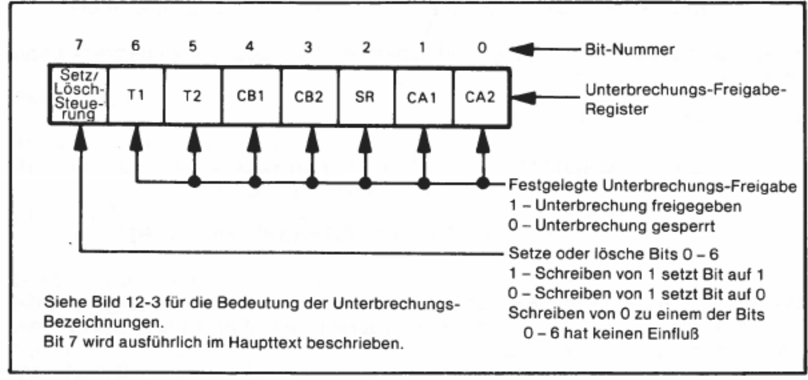


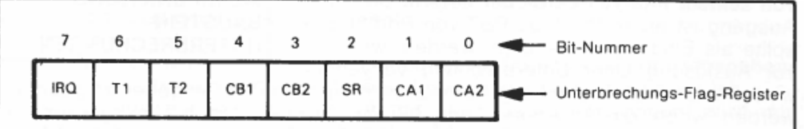
Bild 12-2. Beschreibung der Unterbrechungs-Freigabe-Register des VIA 6522.

2) Gib CB1- und CB2-Unterbrechung frei, sperre alle übrigen, lösche CB1- und CB2-Flags.

```
LDA   #%01100111  ;SPERRE ALLE ÜBRIGEN
                  ; UNTERBRECHUNGEN
STA   VIAIER
LDA   #%10011000
STA   VIAIFR      ;LÖSCHE CB1-, CB2- UNTERBRECHUNGS-
                  ; FLAGS
STA   VIAIER      ;GIB CB1-, CB2-UNTERBRECHUNGEN FREI
```

Beachten Sie, daß Bit 7 des Unterbrechungsflag-Registers und Bit 7 des Unterbrechungsfreigabe-Registers beide sehr speziell sind. Bit 7 des Unterbrechungsflag-Registers zeigt den Status des $\overline{IRQ}$-Ausganges an – das heißt, es ist 1, wenn irgendeine der Unterbrechungen sowohl aktiv wie freigegeben ist. **Bit 7 des Unterbrechungsfreigabe-Registers ist die Setz/Lösch-Steuerung,** die früher erwähnt wurde. Beachten Sie, daß Bit 7 des Unterbrechungsflag-

Registers nicht direkt gelöscht werden kann. Es kann nur entweder durch Löschen aller aktiven Unterbrechungs-Flags oder durch Sperren aller aktiven Unterbrechungen gelöscht werden.



Bit 7 zeigt den Status des $\overline{IRQ}$-Ausgangs an. Dieses Bit entspricht der folgenden Logik-Funktion:

$IRQ = (IFR6 \wedge IER6) \vee (IFR5 \wedge IER5) \vee (IFR4 \wedge IER4) \vee (IFR3 \wedge IER3) \vee (IFR2 \wedge IER2) \vee (IFR1 \wedge IER1) \vee (IFR0 \wedge IER0)$

Die Bits 0 – 6 sind Zwischenspeicher, die wie folgt gesetzt und gelöscht werden:

Bit-Nr.	Gesetzt durch	Gelöscht durch
0	Aktiver Übergang des Signals am Pin CA2	Lesen oder Schreiben des A-Port-Ausgangsregister (ORA) unter Verwendung der Adresse 0001.
1	Aktiver Übergang des Signals am Pin CA1	Lesen oder Schreiben des A-Port-Ausgangsregisters (ORA) unter Verwendung der Adresse 0001
2	Abschluß von acht Verschiebungen	Lesen oder Schreiben des Schieberegisters
3	Aktiver Übergang des Signals am Pin CB2	Lesen oder Schreiben des B-Port-Ausgangsregisters
4	Aktiver Übergang des Signals am Pin CB1	Lesen oder Schreiben des B-Port-Ausgangsregisters
5	Ablauf von Zeitgeber 2	Lesen des niederwertigen Zählers von T2 oder Schreiben des hochwertigen Zählers von T2
6	Ablauf von Zeitgeber 1	Lesen des niederwertigen Zählers von T1 oder Schreiben des hochwertigen Zwischenspeichers von T2

Bild 12-3. Beschreibung des Unterbrechungs-Flag-Registers des VIA 6522.

Beachten Sie folgendes über VIA-Unterbrechungen:

1) **Die VIA-Unterbrechungs-Freigabebits haben die entgegengesetzte Polarität des I- (oder Unterbrechungssperr-) Flags der 6502.** Das heißt, sie müssen 1 sein, um eine Unterbrechung freizugeben.
2) **RESET sperrt alle Unterbrechungen.**
3) **Die CPU kann Bit 7 des Unterbrechungsflag-Registers prüfen um festzustellen, ob irgendwelche Unterbrechungen sowohl aktiv wie freigegeben sind.** Dieses Bit wird gesetzt bleiben, bis keine Unterbrechung sowohl aktiv wie freigegeben ist.
4) **Der VIA wird sich an eine Unterbrechung erinnern, die auftritt, wenn VIA-Unterbrechungen gesperrt sind, und wird eine Anforderung über $\overline{IRQ}$ ausgeben, wenn der VIA freigegeben wird.**

Über VIA-Unterbrechungen werden mehrere Beispiele später in diesem Kapitel behandelt.

Unterbrechungen mit Mehrfunktions-Bausteinen 6530 und 6532

6530 UND 6532 MEHRFUNKTIONS-BAUSTEIN-UNTERBRECHUNGEN

Der Baustein 6530 kann eine Unterbrechung von seinem Intervall-Zeitgeber liefern. Der IRQ-Ausgang ist auch Anschluß PB7 von Port B und sollte als Eingang eingerichtet werden, wenn er zur Auslösung einer Unterbrechung verwendet wird. Die Unterbrechung kann durch Einschreiben in den Zeitgeber freigegeben werden, wobei die Adressenleitung A3 High ist. Die Unterbrechung kann gesperrt werden, indem zum Zeitgeber mit Low auf der Adressenleitung A3 geschrieben wird. Sie kann gelöscht werden, entweder durch Lesen oder durch Schreiben des Zeitgebers, nachdem eine Unterbrechung aufgetreten ist.

Der Baustein 6532 kann eine Zeitgeber-Unterbrechung wie der Baustein 6530 liefern. Er kann auch eine Unterbrechung liefern, die auf dem Auftreten einer Flanke an PA7 basiert. PA7 arbeitet daher ähnlich wie CA1 oder CB1 auf einem PIA 6520 oder VIA 6522. Die Unterbrechung kann entweder bei einem Low-zu-High-Übergang (positive Flanke) oder einem High-zu-Low-Übergang (negative Flanke) auftreten.

6532-Unterbrechungen werden durch Schreiben zu und Lesen von spezifizierten Adressen gesteuert und geprüft (siehe Tabelle 12-2 für eine Beschreibung der Adressen in einem 6532-Baustein). Beachten Sie folgendes:

1) Zur Steuerung der PA7-Unterbrechung schreiben Sie einfach irgendwelche Daten in den E/A-Abschnitt des 6532, gegeben durch:
   RS = 1, um E/A zu aktivieren, anstatt des auf der Platine befindlichen RAM's.
   A2 = 1, A4 = 0

Die beiden niedrigstwertigen Adressen-Bits (nicht die Daten) steuern dann die PA7-Betriebsart wie folgt:

   A1 = 1, um die PA7-Unterbrechung freizugeben, 0 um sie zu sperren.
   A0 = 1 für eine positive (Low-zu-High) Flanken-Feststellung, 0 für eine negative (High-zu-Low) Flanke.

2) Um die Unterbrechungs-Flags zu lesen und zu löschen, lesen Sie von der Adresse im E/A-Abschnitt des 6532, gegeben durch:
   RS = 1 zum Aktivieren von E/A, anstatt des auf der Platine befindlichen RAM's.
   A2 = 1, A0 = 1

Bit 7 ist das Zeitgeber-Unterbrechungsflag und Bit 6 das PA7-Unterbrechungsflag. Dies kann leicht mittels des Befehls Bit-Test gelesen werden (Bit 7 wird zum Vorzeichen-Flag und Bit 6 zum Überlauf-Flag transferiert).

ACIA-Unterbrechungen

6850-ACIA-UNTERBRECHUNGEN

Der ACIA 6850 kann ebenfalls als Quelle für Unterbrechungen dienen. Sie sollten folgende Eigenschaften des ACIA bei einem auf Unterbrechungen basierenden System beachten:

1) **Die Sende-Unterbrechung (ACIA ist bereit für Daten) wird nur freigegeben, wenn das Steuerregister-Bit 6 = 0 und das Steuerregister-Bit 5 = 1 ist.**
2) **Empfänger-Unterbrechung (ACIA hat neue Daten empfangen) wird nur freigegeben, wenn das Steuerregister-Bit 7 = 1 ist.**
3) **Das Haupt-Reset beeinflußt die Unterbrechungs-Freigabe-Bits nicht.**
4) **Bit 7 des Statusregisters wird gesetzt, wenn eine Unterbrechung aufgetreten ist.** Dieses Bit kann entweder durch Lesen von Daten vom ACIA oder durch Schreiben von Daten in den ACIA gelöscht werden.

Tabelle 12-2. Adressieren des Mehrfunktions-Bausteins 6532.

Auswahl-Leitungen							Adressier-Art
$\overline{RS}$	R/$\overline{W}$	A4	A3	A2	A1	A0	
							RAM-Adressierung
0	1(0)	X	X	X	X	X	Lies (Schreib) RAM. A0 – A6 wählt RAM-Adresse
							E/A-Adressierung
1	1(0)	X	X	0	0	0	Lies (Schreibe) Port-A-Daten
1	1(0)	X	X	0	0	1	Lies (Schreibe) Port-A-Datenrichtungs-Register
1	1(0)	X	X	0	1	0	Lies (Schreibe) Port-B-Daten
1	1(0)	X	X	0	1	1	Lies (Schreibe) Port-B-Datenrichtungs-Register
							Flanken-Feststellungs-Steuerung
1	0	0	X	1	0	X	Sperre Unterbrechung von PA7
1	0	0	X	1	1	X	Gib Unterbrechung von PA7 frei
1	0	0	X	1	X	0	Feststellung der negativen Flanke
1	0	0	X	1	X	1	Feststellung der positiven Flanke
1	1	X	X	1	X	1	**Lies und lösche Unterbrechungs-Flags** Bit 7 ist das Zeitgeber-Flag Bit 6 ist das PA7-Flag
							Schreibe Zählung zum Intervall-Zeitgeber
1	0	1	0	1	X	X	und sperre Zeitgeber-Unterbrechung
1	0	1	1	1	X	X	und gib Zeitgeber-Unterbrechung frei
1	0	1	X	1	0	0	und dekrementiere bei jedem ϕ 2-Impuls
1	0	1	X	1	0	1	und dekrementiere alle 8 ϕ 2-Impulse
1	0	1	X	1	1	0	und dekrementiere alle 64 ϕ 2-Impulse
1	0	1	X	1	1	1	und dekrementiere alle 1024 ϕ 2-Impulse

Für alle Operationen CS1 = 1, $\overline{CS2}$ = 0.
Logik-Pegel: 0 bedeutet Low-Pegel
1 bedeutet High-Pegel
X bedeutet, daß der Pegel dieses Signals ohne Bedeutung ist (entweder 0 oder 1)

6502-Abfrage-Unterbrechungssysteme

ABFRAGE-UNTERBRECHUNGEN

Die meisten Unterbrechungs-Systeme des 6502 müssen entweder den PIA, VIA, ACIA oder andere Bausteine abfragen um zu bestimmen, welcher eine Unterbrechung bewirkt hat. Die Abfrage- (Polling-) Methode lautet:

1) Teste jeden PIA durch Prüfen der Steuerregister-Bits 6 und 7:

```
BIT   PIACR    ;PRÜFE PIA-STATUSBITS
BMI   INT1     ;VERZWEIGE ZUR UNTERBRECHUNG 1,
               ; WENN BIT 7 GESETZT
BVS   INT2     ;VERZWEIGE ZU UNTERBRECHUNG 2,
               ; WENN BIT 6 GESETZT
```

2) Teste jeden VIA durch Prüfen des Unterbrechungsflag-Register-Bit 7:

```
BIT   VIAFR    ;SIND IRGENDWELCHE UNTERBRECHUN-
               ; GEN AN DIESEM VIA AKTIV?
BMI   INTV     ;JA, PRÜFE ALLE FLAGREGISTER
```

Sie müssen noch das Unterbrechungsflag-Register prüfen, wenn es mehr als eine mögliche Unterbrechungsquelle von einem speziellen VIA gibt. Alles, was Ihnen Bit 7 sagt, besteht darin, daß es wenigstens eine Quelle gibt, die sowohl aktiv wie freigegeben ist.

3) Teste jeden ACIA durch Prüfen des Statusregister-Bits 7:

```
BIT   ACIASR   ;SIND IRGENDWELCHE UNTERBRECHUN-
               ; GEN AKTIV IN DIESEM ACIA?
BMI   INTA     JA, BESTIMME WELCHE ERFORDERLICH IST
```

Die Unterbrechung könnte noch entweder eine Empfänger- oder Sende-Unterbrechung sein.

Die wichtigsten Eigenschaften eines Abfrage-Systems des 6502 sind:

1) **Die erste geprüfte Unterbrechung besitzt die höchste Priorität, da die verbleibenden Unterbrechungen nicht geprüft werden, wenn die erste aktiv ist.** Die zweite Unterbrechung besitzt die nächsthöhere Priorität usw.

2) **Die Service-Routine muß die Unterbrechungs-Flags von den PIAs, VIAs, ACIAs oder anderen Bausteinen löschen, wenn dieses Löschen nicht automatisch ausgeführt wird.**

Der Programmierer sollte besonders sorgfältig sein bei:

- **Verwendung von PIAs als Unterbrechungs-Ausgabeports.** Ein "Dummy Read" des Ports ist erforderlich, da das Unterbrechungs-Flag nicht automatisch gelöscht wird, wenn Daten in den Port geschrieben werden. PIA-Status- (Unterbrechungs-) Flags werden nur gelöscht, wenn die Datenregister gelesen werden.
- **Verwendung von VIAs in der unabhängigen Betriebsart oder über Adressen, die die Unterbrechungs-Flags nicht beeinflussen.**

Das Unterbrechungs-Flag muß dann ausdrücklich gelöscht werden, indem eine logische 1 in das entsprechende Bit des Unterbrechungsflag-Registers geschrieben wird.

NACHTEILE DER ABFRAGE-UNTERBRECHUNGEN

Abfrage-Routinen sind ausreichend, wenn es nur einige Eingänge gibt. Sind jedoch mehrere Eingänge vorhanden, so sind die Abfrage-Routinen langsam und mühsam aus folgenden Gründen:

1) Die durchschnittliche Anzahl der Abfrage-Operationen steigt linear mit der Anzahl der Eingänge. Im Durchschnitt wird man natürlich nur die Hälfte der Eingänge abzufragen haben, bevor man den richtigen findet. Man kann die durchschnittliche Anzahl der Abfrage-Operationen etwas verringern, indem man die am häufigsten verwendeten Eingänge zuerst prüft.
2) PIA-, VIA- und ACIA-Adressen sind selten aufeinanderfolgend oder im gleichen Abstand. Daher sind separate Befehle erforderlich, um jeden Eingang zu prüfen. Abfrage-Routinen sind deshalb schwierig zu erweitern. Tabelle von E/A-Adressen könnten durch Plazieren der Basis-Adresse auf Seite Null verwendet werden, sowie durch Verwendung der nach-indizierten Adressierung durch Plazieren der gesamten Tabelle auf Seite Null und Verwendung der vor-indizierten Adressierung.
3) Unterbrechungen, die zuerst abgefragt werden, können jene sperren, die später abgefragt werden, außer die Reihenfolge der Abfrage wird variiert. Jedoch das Fehlen aufeinanderfolgender Adressen macht ein Variieren der Reihenfolge der Abfrage sehr schwierig.

Vektorisierte Unterbrechungs-Systeme mit dem 6502

6502 VEKTORISIERTE UNTERBRECHUNGEN

Das Problem des Abfragens in Systemen mit dem 6502 wurde durch spezielle Verfahren für bestimmte Anwendungen oder Mikrocomputer gelöst. **Beachten Sie, daß es keinen Weg gibt um festzustellen, daß der 6502 eine Unterbrechung angenommen hat, außer durch Beobachten, wann die Adressen FFFE und FFFF auf dem Adressenbus aufscheinen. Spezielle Hardware kann dann den Vektor ersetzen, der durch die tatsächliche Quelle geliefert wird[4].** Wir wollen das vektorisierte Unterbrechungs-System des 6502 nicht weiter besprechen.

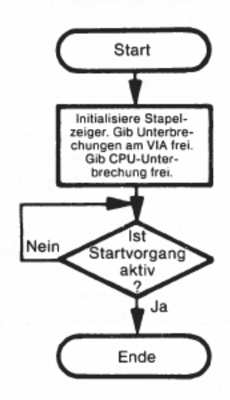
BEISPIELE

Eine Start-Unterbrechung

Zweck: Der Computer wartet auf das Auftreten einer VIA-Unterbrechung, bevor er mit tatsächlichen Operationen beginnt.

Zahlreiche Systeme verbleiben inaktiv, bis sie der Bedienende tatsächlich startet oder ein DATA-READY-Signal empfangen wird. Bei RESET müssen derartige Systeme den Stapelzeiger initialisieren, die Start-Unterbrechung freigeben und eine endlose Schleife oder Jump-to-self-Befehl ausführen. Erinnern Sie sich daran, daß RESET die Prozessor-Unterbrechung sperrt (indem I auf 1 gesetzt wird) sowie alle VIA-Unterbrechungen (durch Löschen aller VIA-Unterbrechungs-Freigabe-Bits). Im Flußdiagramm ist die Entscheidung, ob der Start aktiv ist, durch die Hardware ausgeführt (d.h., indem die CPU den Unterbrechungs-Eingang intern prüft) anstatt durch Software.

Flußdiagramm:



Quellprogramm:

Hauptprogramm:

```
        LDX     #$FF          ;LEGE STAPEL AN DAS ENDE VON SEITE 1
        TXS
        LDA     #0
        STA     VIAPCR        ;MACHE ALLE STEUERLEITUNGEN ZU
                              ;  EINGÄNGEN
        LDA     #%10000010
        STA     VIAIFR        ;LÖSCHE CA1-UNTERBRECHUNGSFLAG
        STA     VIAIFR        ;GIB CA1-UNTERBRECHUNG FREI
        CLI                   ;GIB CPU-UNTERBRECHUNG FREI
HERE    JMP     HERE          ;WARTE WEITER
```

Unterbrechungs-Service-Routine:

```
        *=INTRP
        LDA     #%10000010
        STA     VIAIFR        ;LÖSCHE CA1-UNTERBRECHUNGSFLAG
        LDX     #$FF          ;REINITIALISIERE STAPELZEIGER
        TXS
        JMP     START
```


Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
Hauptprogramm:				
0000	A2		LDX	#$FF
0001	FF			
0002	9A		TXS	
0003	A9		LDA	#0
0004	00			
0005	8D		STA	VIAPCR
0006 } 0007 }	VIAPCR			
0008	A9		LDA	#%10000010
0009	82			
000A	8D		STA	VIAIFR
000B } 000C }	VIAIFR			
000D	8D		STA	VIAIER
000E } 000F }	VIAIER			
0010	58		CLI	
0011	4C	HERE	JMP	HERE
0012	11			
0013	00			
Unterbrechungs-Service-Routine:				
INTRP	A9		LDA	#%10000010
INTRP+1	82			
INTRP+2	8D		STA	VIAIFR
INTRP+3 } INTRP+4 }	VIAIFR			
INTRP+5	A2		LDX	#$FF
INTRP+6	FF			
INTRP+7	9A		TXS	
INTRP+8	4C		JMP	START
INTRP+9 } INTRP+A }	START			

UNTERBRECHUNGEN BEI SPEZIELLEN MIKROCOMPUTERN

Die genaue Lage der Unterbrechungs-Service-Routine variiert mit dem Mikrocomputer. Wenn Ihr Mikrocomputer keinen Monitor besitzt, kann man eine beliebige Adresse in die Speicherplätze FFFE und FFFF plazieren (oder Speicherplätze, die diesen Adressen entsprechen). Man muß dann die Unterbrechungs-Routine an den Adressen starten, die man auswählt. Natürlich sollten Sie die Routine so plazieren, daß sie nicht feste Adressen oder andere Programme stört.

HANDHABUNG VON UNTERBRECHUNGEN DURCH MONITORE

Wenn Ihr Mikrocomputer-System ein Monitorprogramm besitzt, so wird der Monitor die Adressen FFFE und FFFF belegen. Diese Adressen werden entweder eine Start-Adresse enthalten, in die Sie Ihre Unterbrechungs-Service-Routine plazieren müssen, oder sie werden die Start-Adresse einer Routine enthalten, die Ihnen die Wahl der Start-Adresse der Unterbrechungs-Routine gestattet. Eine typische Monitor-Routine wäre:

```
MONINT  JMP    (USRINT)     ;SPRINGE ZU ANWENDER-ADRESSE FÜR
                            ;  UNTERBRECHUNG
```

Sie müssen dann die Adresse Ihrer Service-Routine in die Speicherplätze USRINT und USRINT+1 plazieren. Erinnern Sie sich daran, daß MONINT eine Adresse im Monitorprogramm ist und ihr Wert in den Adressen FFFE und FFFF liegt.

Sie können das Laden der Speicherplätze USRINT und USRINT+1 in Ihr Hauptprogramm einschließen:

```
        LDA    #USRL        ;LADE LSB'S DER ANWENDER-UNTER-
                            ;  BRECHUNGSADRESSE
        STA    USRINT
        LDA    #USRM        ;LADE MSB'S DER ANWENDER-UNTER-
                            ;  BRECHUNGSADRESSE
        STA    USRINT+1
```

Diese Befehle müssen der Freigabe der Unterbrechungen vorausgehen.

Das Hauptprogramm gibt nur die Unterbrechung vom Start-VIA frei. Wir haben angenommen, daß die Startleitung mit dem Eingang CA1 des VIA verbunden ist und daß die aktive Flanke die abfallende ist (d.h. ein High-zu-Low-Übergang). Andere Konfigurationen würden einfach andere Werte im VIA-Peripherie-Steuerregister erfordern.

Beachten Sie, daß die VIA-Unterbrechung freigegeben und der Stapelzeiger geladen wird, bevor die CPU-Unterbrechung freigegeben wird (durch Löschen des I-Bits). Was würde geschehen, wenn Sie das I-Bit vor dem Laden des Stapelzeigers löschen würden? Dies wäre jedoch kein wesentliches Problem, wenn der Monitor bereits einen Wert in den Stapelzeiger plaziert.

In diesem Beispiel ist die Rückkehr-Adresse und Statusregister, die der 6502 bei Annahme einer Unterbrechung in den Stapel speichert, nicht sehr nützlich. Daher reinitialisiert die Service-Routine einfach den Stapelzeiger.

Beachten Sie, daß wir den Befehl JMP HERE durch eine bedingte Verzweigung ersetzen könnten, die einen gesicherten Sprung liefert, wie etwa BNE HERE. Das Null-Flag ist nicht null, da die letzte Operation die eine war, die die CA1-Unterbrechung freigab. Diese Kurzform ist häufig sehr nützlich infolge der Tatsache, daß der 6502 keinen unbedingten Sprung mit relativer Adressierung besitzt.

Erinnern Sie sich daran, daß $\overline{\text{RESET}}$ und das Annehmen einer Unterbrechung automatisch das Unterbrechungssystem sperrt. Dies gestattet der Start-Routine das Konfigurieren aller VIAs und anderer Unterbrechungsquellen, ohne selbst unterbrochen zu werden. Beachten Sie, daß Sie das CA1-Unterbrechungsflag ausdrücklich löschen müssen, oder es würde wieder unterbrechen, sobald das Unterbrechungssystem wieder freigegeben wird. Sie könnten auch das Flag durch Lesen des VIA-Ausgangsregisters A von der Quittierungs-Adresse löschen (siehe Tabelle 11-7).

Eine Tastatur-Unterbrechung

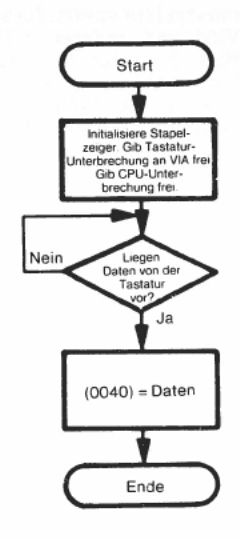
TASTATUR-UNTERBRECHUNG

Zweck: Der Computer wartet auf eine Tastatur-Unterbrechung und plaziert die Daten von der Tastatur in den Speicherplatz 0040.

Beispiel:

Tastatur-Daten = 06
Ergebnis: (0040) = 06

Flußdiagramm:



Quellprogramm:

```
Hauptprogramm:
        LDX   #$FF          ;BRINGE STAPEL AN DAS ENDE VON
                            ;   SEITE 1
        TXS
        LDA   #0
        STA   VIAPCR        ;MACHE ALLE STEUERLEITUNGEN ZU
                            ;   EINGÄNGEN
        STA   VIADDRA       ;MACHE PORT-A-LEITUNGEN ZU
                            ;   EINGÄNGEN
        LDA   #%10000010
        STA   VIAIFR        ;LÖSCHE TASTATUR-UNTERBRECHUNGS-
                            ;   FLAG
        STA   VIAIER        ;GIB TASTATUR-UNTERBRECHUNG VON VIA
                            ;   FREI
        CLI                 ;GIB CPU-UNTERBRECHUNG FREI
HERE    JMP   HERE          ;"DUMMY"-HAUPTPROGRAMM
```



```
Unterbrechungs-Service-Routine:
        *=INTRP
        PHA                 ;BEWAHRE AKKUMULATOR IM STAPEL AUF
        LDA   VIAORA        ;HOLE TASTATUR-DATEN
        STA   $40           ;BEWAHRE TASTATUR-DATEN AUF
        PLA                 ;SPEICHERE AKKUMULATOR VON STAPEL
                            ;   ZURÜCK
        RTI
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
Hauptprogramm:				
0000	A2		LDX	#$FF
0001	FF			
0002	9A		TXS	
0003	A9		LDA	#0
0004	00			
0005	8D		STA	VIAPCR
0006 } 0007 }	VIAPCR			
0008	8D		STA	VIADDRA
0009 } 000A }	VIADDRA			
000B	A9		LDA	#%10000010
000C	82			
000D	8D		STA	VIAIFR
000E } 000F }	VIAIFR			
0010	8D		STA	VIAIER
0011 } 0012 }	VIAIER			
0013	58		CLI	
0014	4C	HERE	JMP	HERE
0015	14			
0016	00			
Unterbrechungs-Service-Routine:				
INTRP	48		PHA	
INTRP+1	AD		LDA	VIAORA
INTRP+2 } INTRP+3 }	VIAORA			
INTRP+4	85		STA	$40
INTRP+5	40			
INTRP+6	68		PLA	
INTRP+7	40		RTI	

Sie müssen den VIA vollständig vor der Freigabe der Unterbrechungen konfigurieren. Dies beinhaltet das Einrichten der Richtungen der Ports, Bestimmen der Übergänge, die auf den Tastleitungen erkannt werden müssen, und die Freigabe von Zwischenspeichern (erinnern Sie sich daran, daß das Setzen von Bit 0 des Hilfs-Steuerregisters den Zwischenspeicher des Ports A freigibt).

JMP HERE ist ein Befehl für eine endlose Schleife (Jump-to-self), die zur Darstellung des Hauptprogramms verwendet wird. Nachdem Unterbrechungen in einem Arbeitssystem freigegeben wurden, führt das Hauptprogramm seine eigentliche Arbeit aus, bis eine Unterbrechung auftritt, und nimmt dann die Ausführung nach Abschluß der Unterbrechungs-Service-Routine wieder auf.

ÄNDERN DER UNTERBRECHUNGS-RÜCKKEHR-ADRESSE

Der RTI-Befehl am Ende der Service-Routine transferiert die Steuerung zurück zum JMP-Befehl im Hauptprogramm. Wenn Sie dies vermeiden wollen, können Sie einfach den Befehlszähler im Stapel ändern. Erinnern Sie sich daran, daß der Stapel immer auf Seite 1 (Adressen 0100 –01FF) liegt, der Stapelzeiger die Adresse des nächsten leeren Speicherplatzes enthält und die Reaktion auf die Unterbrechung den Befehlszähler in den Stapel unterhalb des Stapelregisters plaziert. **Daher wird das folgende Programm den Befehlszähler im Stapel inkrementieren, ohne ihn hierbei zu entfernen.**

```
        TXS                 ;MACHE STAPELZEIGER ZU INDEX
        INC     $0102,X     ;INKREMENTIERE LSB'S DER
                            ;  RÜCKKEHR-ADRESSE
        BNE     DONE
        INC     $0103,X     ;UND ÜBERTRAG ZU MSB'S FALLS
                            ;  ERFORDERLICH
DONE    (nächster Befehl)
```

Da der 6502 seine Register nicht automatisch aufbewahrt (anders als das Statusregister), können Sie ihn zum Übertragen von Parametern und Ergebnissen zwischen dem Hauptprogramm und der Unterbrechungs-Service-Routine verwenden. So könnten Sie daher die Daten im Akkumulator lassen, anstatt im Speicherplatz 0040. Dies ist jedoch eine gefährliche Praxis, die man vermeiden sollte, mit Ausnahme in extrem einfachen Systemen. In den meisten Anwendungen verwendet der Prozessor seine Register während der normalen Programm-Ausführung. Wenn nun die Unterbrechungs-Service-Routine zufällig den Inhalt dieser Register ändert, gibt es unübersehbare Probleme. **Im allgemeinen sollte keine Unterbrechungs-Service-Routine jemals irgendwelche Register ändern, außer der Inhalt dieser Register wurde vor seiner Änderung aufbewahrt und nach Abschluß der Routine zurückgespeichert.**

Beachten Sie, daß Sie die Unterbrechungen am Ende der Service-Routine nicht ausdrücklich wieder freigeben müssen. Der Grund hierfür ist, daß der RTI-Befehl automatisch das alte Statusregister (P) mit dem Unterbrechungs-Freigabebit in seinen ursprünglichen Zustand zurückversetzt. In der Tat werden Sie das Unterbrechungs-Freigabebit im Stapel (Bit 2 der obersten Speicherstelle) ändern müssen, wenn Sie nicht wollen, daß die Unterbrechungen wieder freigegeben werden.

Die Verwendung des Stapels ist die allgemeinste Lösung zum Aufbewahren und Zurückspeichern von Registern. Der Befehl PHA bewahrt den Inhalt des Akkumulators im Stapel auf, und der Befehl PLA speichert den Inhalt des Akkumulators vom Stapel zurück. Dieses Verfahren kann unbegrenzt erweitert werden (solange es Platz im Stapel gibt), da verschachtelte Service-Routinen nicht die Daten zerstören werden, die in früheren Routinen aufbewahrt wurden.

Sie können alle Register im Stapel aufbewahren (erinnern Sie sich daran, daß der Status automatisch gerettet wird) nach einer Unterbrechung mit der Sequenz:

```
        PHA                 ;BEWAHRE AKKUMULATOR AUF
        TXA                 ;BEWAHRE INDEXREGISTER X AUF
        PHA
        TYA                 ;BEWAHRE INDEXREGISTER Y AUF
        PHA
```

Beachten Sie, daß es keinen direkten Weg zum Transferieren von Daten zwischen dem Stapel und den Indexregistern gibt. Der Inhalt des Akkumulators muß zuerst gerettet werden (weshalb?).

Sie können die Register vom Stapel zurückspeichern (erinnern Sie sich daran, daß RTI automatisch den Status nach einer Unterbrechungs-Service-Routine zurückspeichert) indem Sie die Daten vom Stapel in der entgegengesetzten Reihenfolge entfernen, in der sie eingegeben wurden.

```
        PLA                 ;SPEICHERE INDEXREGISTER Y ZURÜCK
        TAY
        PLA                 ;SPEICHERE INDEXREGISTER Y ZURÜCK
        TAX
        PLA                 ;SPEICHERE AKKUMULATOR ZURÜCK
```

Beachten Sie, daß der Akkumulator zuerst gerettet und zuletzt zurückgespeichert wird.

Eine alternative Lösung wäre, für die Unterbrechungs-Routine die Steuerung zu behalten, bis eine gesamte Textzeile empfangen wurde (zum Beispiel eine Reihe von Zeichen, endend mit einem Wagenrücklauf). Das Hauptprogramm wäre:

FÜLLEN EINES PUFFERS ÜBER UNTERBRECHUNGEN

Hauptprogramm:

```
        LDX     #$FF        ;BRINGE STAPEL AN DAS ENDE
                            ; VON SEITE 1
        TXS
        LDA     #0
        STA     VIAPCR      ;MACHE ALLE STEUERLEITUNGEN ZU
                            ; EINGÄNGEN
        STA     VIADDRA     ;MACHE PORT-A-LEITUNGEN ZU
                            ; EINGÄNGEN
        STA     $40         ;LÖSCHE PUFFER-INDEX ZU BEGINN
        LDA     #%10000010
        STA     VIAIFR      ;LÖSCHE TASTATUR-UNTERBRECHUNGS-
                            ; FLAG
        STA     VIAIER      ;GIB TASTATUR-UNTERBRECHUNG VON VIA
                            ; FREI
        CLI                 ;GIB CPU-UNTERBRECHUNG FREI
HERE    JMP     HERE        ;"DUMMY"-HAUPTPROGRAMM
```

Unterbrechungs-Service-Routine:

```
        *=INTRP
        PHA                 ;BEWAHRE AKKUMULATORIM STAPEL AUF
        TXA                 ;BEWAHRE INDEXREGISTER X IM STAPEL
                            ; AUF
        PHA
        LDX     $40         ;HOLE PUFFER-INDEX
        LDA     VIAORA      ;HOLE TASTATUR-DATEN
        STA     $41,X       ;BEWAHRE DATEN IN PUFFER AUF
        CMP     #CR         ;SIND DATEN EIN WAGENRÜCKLAUF?
        BEQ     ENDL        ;JA, ENDE DER ZEILE
        INC     $40         ;NEIN, INKREMENTIERE PUFFER-ZEIGER
        PLA                 ;SPEICHERE INDEXREGISTER X VOM
                            ; STAPEL ZURÜCK
        TAX
        PLA                 ;SPEICHERE AKKUMULATOR VOM STAPEL
                            ; ZURÜCK
        RTI
ENDL    JMP     LPROC       ;SETZE OHNE UNTERBRECHUNGEN FORT
```

Dieses Programm füllt einen Puffer, beginnend beim Speicherplatz 0041, bis es ein Wagenrücklauf-Zeichen (CR) empfängt. Der Speicherplatz 0040 beinhaltet den momentanen Puffer-Index.
Wenn der Prozessor einen Wagenrücklauf empfängt, läßt er das Unterbrechungssystem gesperrt, während er diese Zeile verarbeitet.

Eine alternative Lösung wäre das Füllen eines anderen Puffers, während das erste bearbeitet wird. Diese Lösung wird "doppelte Pufferung" genannt.

VERDOPPELN DES PUFFERNS

Die Verarbeitungs-Routine für die Zeile hat bei der Adresse LPROC mit gesperrten Unterbrechungen begonnen und mit dem ursprünglichen Inhalt der Register (P, A und X) sowie der Rückkehr-Adresse im Stapel.
In einer realen Anwendung könnte die CPU andere Aufgaben zwischen den Unterbrechungen ausführen. Sie könnte zum Beispiel eine Zeile editieren oder von einem Puffer zu einem anderen übertragen, während die Unterbrechung einen weiteren Puffer füllt.

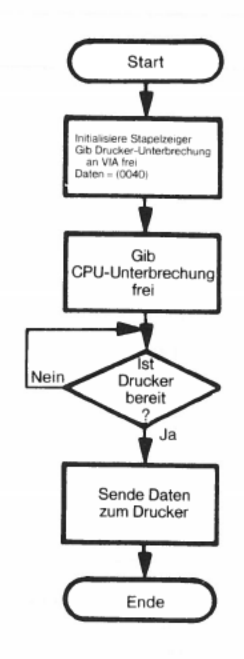
Eine Drucker-Unterbrechung

Zweck: Der Computer wartet auf eine Drucker-Unterbrechung und sendet die Daten vom Speicherplatz 0040 zum Drucker.

Beispiel:

(0040) = 51_{16}
Ergebnis: Der Drucker empfängt ein 51_{16} (ASCII Q), wenn er bereit ist.

Flußdiagramm:



Quell-Programm:

Hauptprogramm:

```
        LDX     #$FF        ;LEGE STAPEL AN DAS ENDE VON SEITE 1
        TXS
        STX     VIADDRB     ;MACHE PORT-B-LEITUNGEN ZU
                            ; AUSGÄNGEN
        LDA     #0
        STA     VIAPCR      ;MACHE ALLE STEUERLEITUNGEN ZU
                            ; EINGÄNGEN
        LDA     #%10000010
```

```
         STA    VIAIFR           ;LÖSCHE DRUCKER-UNTERBRECHUNGS-
                                 ;  FLAG
         STA    VIAIER           ;GIB DRUCKER-UNTERBRECHUNG VON VIA
                                 ;  FREI
         CLI                     ;GIB CPU-UNTERBRECHUNGEN FREI
HERE     JMP    HERE             ;"DUMMY"-HAUPTPROGRAMM
```

Unterbrechungs-Service-Routine:

```
         *=INTRP
         PHA                     ;BEWAHRE AKKUMULATOR IN STAPEL AUF
         LDA    $40              ;HOLE DATEN
         STA    VIAORB           ;SENDE DATEN ZU DRUCKER
         PLA                     ;SPEICHERE AKKUMULATOR VON STAPEL
                                 ;  ZURÜCK
         RTI
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
Hauptprogramm:				
0000	A2		LDX	#$FF
0001	FF			
0002	9A		TXS	
0003	8E		STX	VIADDRB
0004 } 0005 }	VIADDRB			
0006	A9		LDA	#0
0007	00			
0008	8D		STA	VIAPCR
0009 } 000A }	VIAPCR			
000B	A9		LDA	#%10000010
000C	82			
000D	8D		STA	VIAIFR
000E } 000F }	VIAIFR			
0010	8D		STA	VIAIER
0011 } 0012 }	VIAIER			
0013	58		CLI	
0014	4C	HERE	JMP	HERE
0015	14			
0016	00			
Unterbrechungs-Service-Routine:				
INTRP	48		PHA	
INTRP+1	A5		LDA	$40
INTRP+2	40			
INTRP+3	8D		STA	VIAORB
INTRP+4 } INTRP+5 }	VIAORB			
INTRP+6	68		PLA	
INTRP+7	40		RTI	

LEEREN EINES PUFFERS MIT UNTERBRECHUNGEN

Hier, so wie bei der Tastatur, könnten wir den Drucker weiter unterbrechen lassen, bis die ganze Textzeile übertragen ist. Das Hauptprogramm und die Service-Routine wären dann:

Hauptprogramm:

```
         LDX    #$FF
         TXS                     ;LEGE STAPEL AN DAS ENDE VON SEITE 1
         STX    VIADDRB          ;MACHE PORT-B-LEITUNGEN ZU
                                 ;  AUSGÄNGEN
         LDA    #0
         STA    VIAPCR           ;MACHE ALLE STEUERLEITUNGEN ZU
                                 ;  EINGÄNGEN
         STA    $40              ;INITIALISIERE PUFFER-INDEX AUF NULL
         LDA    #%10000010
         STA    VIAIFR           ;LÖSCHE DRUCKER-UNTERBRECHUNGS-
                                 ;  FLAG
         STA    VIAIER           ;GIB DRUCKER-UNTERBRECHUNG VON VIA
                                 ;  FREI
         CLI                     ;GIB CPU-UNTERBRECHUNG FREI
HERE     JMP    HERE             ;"DUMMY"-HAUPTPROGRAMM
```

Unterbrechungs-Service-Routine:

```
         *=INTRP
         PHA                     ;BEWAHRE AKKUMULATOR IN STAPEL AUF
         TXA                     ;BEWAHRE INDEXREGISTER X IN STAPEL
                                 ;  AUF
         PHA
         LDX    $40              ;HOLE PUFFER-INDEX
         LDA    $41,X            ;HOLE EIN DATENBYTE VON PUFFER
         STA    VIAORB           ;SENDE DATEN ZUM DRUCKER
         CMP    #CR              ;SIND DATEN EIN WAGENRÜCKLAUF?
         BEQ    ENDL             ;JA, ENDE DER ZEILE
         INC    $40              ;NEIN, INKREMENTIERE PUFFERZEIGER
         PLA                     ;SPEICHERE INDEXREGISTER X VOM
                                 ;  STAPEL ZURÜCK
         TAX
         PLA                     ;SPEICHERE AKKUMULATOR VOM STAPEL
                                 ;  ZURÜCK
         RTI
ENDL     JMP    LCOMP            ;BEARBEITE VOLLSTÄNDIGE ZEILE
```

Wiederum könnte doppelte Pufferung verwendet werden, damit E/A und Verarbeitung zur gleichen Zeit auftritt, ohne jemals die CPU anzuhalten.

Eine Echtzeit-Takt-Unterbrechung[5,6]

ECHT-ZEIT-TAKT

Zweck: Der Computer wartet auf eine Unterbrechung von einem Echtzeit-Takt[5,6].

Ein Echtzeit-Takt liefert einfach eine gleichmäßige Serie von Impulsen. Das Intervall zwischen den Impulsen kann als Zeitreferenz verwendet werden. Echtzeittakt-Unterbrechungen können gezählt werden, um jedes Vielfache des grundlegenden Zeitintervalls zu geben. Ein Echtzeit-Takt kann durch Teilen des CPU-Taktes gebildet werden, durch einen separaten Zeitgeber oder einen programmierbaren Zeitgeber, wie der im VIA 6522 oder in den Mehrfunktions-Bausteinen 6530 oder 6532 (siehe Kapitel 11), oder durch Verwendung externer Quellen, wie etwa der Wechselspannungs-Netzfrequenz.

FREQUENZ EINES ECHT-ZEIT-TAKTES

Beachten Sie die Überlegungen bei der Bestimmung der Frequenz des Echtzeit-Taktes. Eine hohe Frequenz (etwa 10 kHz) gestattet die Erzeugung eines weiten Bereiches von Zeitintervallen mit hoher Genauigkeit. Andererseits kann der Aufwand beim Zählen von Echtzeittakt-Unterbrechungen beträchtlich sein, und die Zählungen werden rasch die Kapazität eines einzelnen 8-Bit-Registers oder Speicherplatzes übersteigen. Die Wahl der Frequenz hängt von der Genauigkeit und den zeitlichen Anforderungen ihrer Anwendung ab. Der Taktgenerator kann natürlich zum Teil aus Hardware bestehen. Ein Zähler könnte die Hochfrequenz-Impulse zählen und den Prozessor nur gelegentlich unterbrechen. Ein Programm muß dann den Zähler lesen, um die Zeit mit hoher Genauigkeit zu messen.

SYNCHRONISATION MIT ECHTZEIT-TAKT

Ein Problem stellt die Synchronisation von Operationen mit dem Echtzeit-Takt dar. Offensichtlich wird hier ein gewisser Einfluß auf die Genauigkeit des Zeitintervalles vorliegen, wenn die CPU die Messung zufällig während der Taktperiode beginnt, anstatt exakt am Anfang zu starten. Einige Möglichkeiten zur Synchronisation von Operationen sind:

1) Starten Sie die CPU und den Takt zusammen. $\overline{\text{RESET}}$ oder eine Start-Unterbrechung können sowohl den Takt wie die CPU starten.
2) Gestatten Sie der CPU, den Takt mittels Programmsteuerung zu starten und zu stoppen.
3) Verwenden Sie einen hochfrequenten Takt, so daß ein Fehler von weniger als einer Taktperiode klein sein wird.
4) Stellen Sie den Takt (indem Sie auf eine Flanke oder Unterbrechung warten) vor dem Starten der Messung.

PRIORITÄT EINER ECHTZEIT-UHR

Eine Echtzeittakt-Unterbrechung sollte eine sehr hohe Priorität besitzen, da die Genauigkeit der Zeitintervalle durch jede Verzögerung beim Bedienen der Unterbrechung beeinflußt wird. Die Praxis besteht gewöhnlich darin, dem Echtzeit-Takt die höchste Unterbrechungs-Priorität zu geben, mit Ausnahme für einen Betriebsspannungs-Ausfall. **Die Unterbrechungs-Service-Routine des Taktes wird im allgemeinen extrem kurz gehalten, so daß sie andere CPU-Aktivitäten nicht stört.**

a) Warten auf einen Echtzeit-Takt

Quellprogramm:

Hauptprogramm:

```
        LDX   #$FF          ;LEGE STAPEL AN ENDE VON SEITE 1
        TXS
        LDA   #0
        STA   VIAPCR        ;MACHE ALLE STEUERLEITUNGEN
                            ; ZU EINGÄNGEN
        LDA   #%10000010
        STA   VIAIFR        ;LÖSCHE TAKTUNTERBRECHUNGS-FLAG
        STA   VIAIER        ;GIB TAKT-UNTERBRECHUNG VON VIA FREI
        CLI                 ;GIB CPU-UNTERBRECHUNG FREI
HERE    JMP   HERE          ;"DUMMY"-HAUPTPROGRAMM
```

Unterbrechungs-Service-Routine:

```
        *=INTRP
        PHA                 ;BEWAHRE AKKUMULATOR IN STAPEL AUF
        LDA   #%10000010
        STA   VIAIFR        ;LÖSCHE TAKTUNTERBRECHUNGS-FLAG
        PLA                 ;SPEICHERE AKKUMULATOR VON STAPEL
                            ; ZURÜCK
        BRK
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
Hauptprogramm:				
0000	A2		LDX	#$FF
0001	FF			
0002	9A		TXS	
0003	A9		LDA	#0
0004	00			
0005	8D		STA	VIAPCR
0006 } 0007 }	VIAPCR			
0008	A9		LDA	#%10000010
0009	82			
000A	8D		STA	VIAIFR
000B } 000C }	VIAIFR			
000D	8D		STA	VIAIER
000E } 000F }	VIAIER			
0010	58		CLI	
0011	4C	HERE	JMP	HERE
0012	11			
0013	00			
Unterbrechungs-Service-Routine:				
INTRP	48		PHA	
INTRP+1	A9		LDA	#%10000010
INTRP+2	82			
INTRP+3	8D		STA	VIAIFR
INTRP+4 } INTRP+5 }	VIAIFR			
INTRP+6	68		PLA	
INTRp+7	00		BRK	

Wenn Bit 0 des VIA-Steuerregisters gleich 0 ist, wird die Unterbrechung an der abfallenden Flanke des Taktes auftreten. Wenn dieses Bit 1 ist, wird die Unterbrechung an der ansteigenden Flanke des Taktes auftreten.

Das Taktunterbrechungs-Flag muß in der Unterbrechungs-Service-Routine ausdrücklich gelöscht werden, da kein E/A-Transfer erforderlich ist. Beachten Sie, daß Port A noch für Daten verwendet werden könnte, solange diese Daten unter Verwendung der Adresse transferiert wurden, die die Unterbrechungsflags nicht beeinflußt (siehe Tabelle 11-7).

Wir könnten natürlich den Impuls selbst unter Verwendung eines der Zeitgeber des 6522 erzeugen. Das folgende Beispiel verwendet Zeitgeber 1 zum Erzeugen eines einzelnen Impulses mit einer Länge von 5000 (1388_{16}) Taktzyklen.

Erinnern Sie sich an folgendes:

1) Die Zähler des Zeitgebers 1 werden von zwei Speicherplätzen (VIAT1L und VIAT1CH) geladen. Das Laden der höchstwertigen Bits der Zeitgeber-Zählung in VIAT1CH startet den Zeitgeber und löscht das T1-Unterbrechungsflag (Bit 6 des Unterbrechungsflag-Registers).
2) Die Betriebsart des Zeitgebers 1 wird durch die Bits 6 und 7 des Hilfs-Steuerregisters bestimmt:
   Bit 6 = 0 für einen einzelnen Impuls und 1 für kontinuierliche Arbeitsweise
   Bit 7 = 0 zum Sperren der Ausgangsimpulse an PB7 und 1 zum Erzeugen derartiger Impulse.
3) Der Abschluß des Zeitgeber-Intervalls setzt das Unterbrechungs-Flag des Zeitgebers 1 (Bit 6 des Unterbrechungsflag-Registers).

Tabelle 11-7 beschreibt die Adressierung des VIA. Bild 11-10 beschreibt das Hilfs-Steuerregister und Bild 12-3 beschreibt das Unterbrechungsflag-Register.

Hauptprogramm:

```
        LDX   #$FF
        TXS               ;LEGE STAPEL AN DAS ENDE VON SEITE 1
        LDA   #0
        STA   VIAACR      ;ERZEUGE EINEN IMPULS
                          ;  VON ZEITGEBER 1
        LDA   #%11000000
        STA   VIAIFR      ;LÖSCHE ZEITGEBER-1-UNTERBRECHUNG
        STA   VIAIER      ;GIB ZEITGEBER-1-UNTERBRECHUNG FREI
        LDA   #$88        ;IMPULSLÄNGE = 5000 (1388 HEX)
        STA   VIAT1L
        LDA   #$13
        STA   VIAT1CH     ;STARTE ZEITGEBER-INTERVALL
        CLI               ;GIB CPU-UNTERBRECHUNG FREI
HERE    JMP   HERE        ;"DUMMY"-HAUPTPROGRAMM
```

Unterbrechungs-Service-Routine:

```
        *=INTRP
        PHA               ;BEWAHRE AKKUMULATOR IN STAPEL AUF
        LDA   #%11000000
        STA   VIAIFR      ;LÖSCHE TAKTUNTERBRECHUNGS-FLAG
        PLA               ;SPEICHERE AKKUMULATOR VOM STAPEL
                          ;  ZURÜCK
        BRK
```

Die einzige Änderung in der Service-Routine ist die Position des Taktunterbrechungs-Flags im Unterbrechungsflag-Register.

b) Warten auf 10 Echtzeittakt-Unterbrechungen

Quellprogramm:

Hauptprogramm:

```
        LDX   #$FF        ;LEGE STAPEL AN DAS ENDE VON SEITE 1
        TXS
        LDA   #0
        STA   VIAPCR      ;MACHE ALLE STEUERLEITUNGEN
                          ;  ZU EINGÄNGEN
        STA   $40         ;LÖSCHE TAKTZÄHLER
        LDA   #%10000010
        STA   VIAIFR      ;LÖSCHE TAKTUNTERBRECHUNGS-FLAG
        STA   VIAIER      ;GIB TAKTUNTERBRECHUNG VON VIA FREI
        LDA   #10         ;ANZAHL DER ZÄHLUNGEN = 10
        CLI               ;GIB CPU-UNTERBRECHUNG FREI
WTTEN   CMP   $40         ;SIND ZEHN ZÄHLUNGEN ABGELAUFEN?
        BNE   WTTEN       ;NEIN, WARTE
        SEI               ;JA, SPERRE CPU-UNTERBRECHUNG
        BRK
```

Unterbrechungs-Service-Routine:

```
        *=INTRP
        PHA               ;BEWAHRE AKKUMULATOR IN STAPEL AUF
        INC   $40         ;INKREMENTIERE TAKTZÄHLER
        LDA   #%10000010
        STA   VIAIFR      ;LÖSCHE TAKTUNTERBRECHUNGS-FLAG
        PLA               ;SPEICHERE AKKUMULATOR VOM STAPEL
                          ;  ZURÜCK
        RTI
```

Natürlich könnten wir die Impulse von dem Zeitgeber des 6522 erzeugen – zum Beispiel könnten wir Zeitgeber 1 in seiner kontinuierlichen Betriebsart verwenden (Bit 6 des Hilfs-Steuerregisters = 1). Die einzige andere Änderung wäre die Bit-Position des Unterbrechungs-Flags.

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
Hauptprogramm:				
0000	A2		LDX	#$FF
0001	FF			
0002	9A		TXS	
0003	A9		LDA	#0
0004	00			
0005	8D		STA	VIAPCR
0006 } 0007 }	VIAPCR			
0008	85		STA	$40
0009	40			
000A	A9		LDA	#%10000010
000B	82			
000C	8D		STA	VIAIFR
000D } 000E }	VIAIFR			
000F	8D		STA	VIAIER
0010 } 0011 }	VIAIER			
0012	A9		LDA	#10
0013	0A			
0014	58		CLI	
0015	C5	WTTEN	CMP	$40
0016	40			
0017	D0		BNE	WTTEN
0018	FC			
0019	78		SEI	
001A	00		BRK	
Unterbrechungs-Service-Routine:				
INTRP	48		PHA	
INTRP+1	E6		INC	$40
INTRP+2	40			
INTRP+3	A9		LDA	#%10000010
INTRP+4	82			
INTRP+5	8D		STA	VIAIFR
INTRP+6 } INTRP+7 }	VIAIFR			
INTRP+8	68		PLA	
INTRP+9	40		RTI	

Diese Unterbrechungs-Service-Routine bringt einfach den Zähler im Speicherplatz 0040 auf den neuesten Stand. Es ist transparent für das Hauptprogramm.

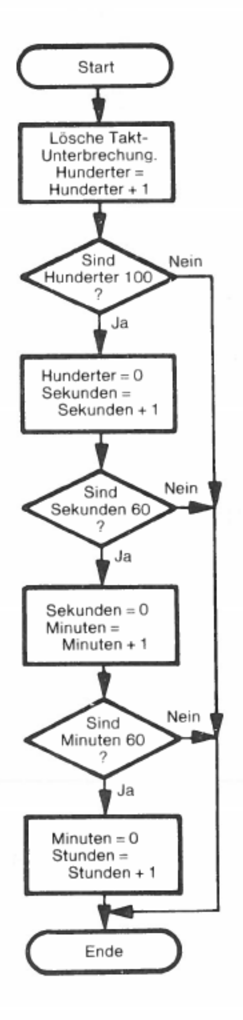
AUFBEWAHRUNG DER ECHTZEIT

Eine realistischere Echtzeit-Unterbrechungs-Routine könnte Echtzeit in mehreren Speicherplätzen aufbewahren. Zum Beispiel verwendet die folgende Routine die Adressen 0040 bis 0043 wie folgt:

0040 – Hundertstel Sekunden
0041 – Sekunden
0042 – Minuten
0043 – Stunden

Wir nehmen an, daß die Routine mit einem Takt von 100 Hz getriggert wird.

Flußdiagramm:



Quellprogramm:

```
        *=INTRP
        PHA                     ;BEWAHRE AKKUMULATOR IN STAPEL
                                AUF
        LDA     #%10000010
        STA     VIAIFR          ;LÖSCHE TAKTUNTERBRECHUNGS-FLAG
        INC     $40             ;BRINGE HUNDERTSTEL SEKUNDEN
                                ; AUF LETZTEN STAND
        LDA     $40
        SEC                     ;GIBT ES EINEN ÜBERTRAG ZU DEN
                                ; SEKUNDEN?
        SBC     #100
        BNE     ENDINT          ;NEIN, DONE
        STA     $40             ;JA, MACHE HUNDERTSTEL NULL
        INC     $41             ;BRINGE SEKUNDEN AUF LETZTEN STAND
        LDA     $41
        SBC     #60             ;GIBT ES EINEN ÜBERTRAG ZU DEN
                                ; MINUTEN?
        BNE     ENDINT          ;NEIN, DONE
        STA     $41             ;JA, MACHE SEKUNDEN NULL
        INC     $42             ;BRINGE MINUTEN AUF LETZTEN STAND
        LDA     $42
        SBC     #60             ;GIBT ES EINEN ÜBERTRAG ZU DEN
                                ; STUNDEN?
        BNE     ENDINT          ;NEIN, DONE
        STA     $42             ;JA, MACHE MINUTEN ZU NULL
        INC     $43             ;BRINGE STUNDEN AUF NEUESTEN
                                STAND
ENDINT  PLA                     ;SPEICHERE AKKUMULATOR VOM STAPEL
                                ; ZURÜCK
        RTI
```

Nun könnte ein Warten von 300 ms im Hauptprogramm mit folgender Routine erzeugt werden:

```
        LDA     $40             ;HOLE MOMENTANE ECHTZEIT
        CLC
        ADC     #30             ;GEWÜNSCHTE ZEIT IST 30 ZÄHLUNGEN
                                ; SPÄTER
        CMP     #100            ;MOD 100
        BCC     WAIT30
        SBC     #100
WAIT30  CMP     $40             ;WARTE BIS ZUR GEWÜNSCHTEN ZEIT
        BNE     WAIT30
```

Wir brauchen nicht ausdrückliche Befehle SET CARRY (SEC), mit Ausnahme in der ersten Operation in der Unterbrechungs-Service-Routine. Die anderen Operationen in der Unterbrechungs-Service-Routine werden nur ausgeführt, wenn die vorhergehende Subtraktion ein Null-Ergebnis erzeugte (und daher einen Übertrag von 1 erzeugte, wodurch ein Borgen angezeigt wird). In dem Warteprogramm wird die Subtraktion überhaupt nur ausgeführt, wenn der Übertrag 1 ist (andernfalls tritt eine Verzweigung auf).

Natürlich könnte das Programm andere Aufgaben ausführen und nur die abgelaufene Zeit gelegentlich prüfen. Wie würden Sie eine Verzögerung von 7 Sekunden erzeugen? 3 Minuten?

Manchmal werden Sie vielleicht wollen, daß die Zeit entweder in BCD-Ziffern oder in ASCII-Zeichen vorliegt. Wie würden Sie das letzte Programm ändern, um diese Alternativen zu handhaben?

SPERREN VON UNTERBRECHUNGEN

Sie können die Takt-Unterbrechung (oder jede andere Unterbrechung) auf eine der folgenden Wege sperren, wenn sie nicht länger benötigt wird:

1) Durch Ausführung eines SEI-Befehls im Hauptprogramm. Dies sperrt das gesamte Unterbrechungs-System. Ein SEI-Befehl in der Service-Routine hat keinen Einfluß, da RTI das alte I-Flag zurückspeichert. Tatsächlich sperrt der 6502 Unterbrechungen automatisch während der Service-Routine.
2) Durch Löschen des entsprechenden Bits des Unterbrechungs-Freigabe-Registers während der Service-Routine oder während des Hauptprogramms. Dies sperrt nur die einzige Unterbrechungs-Quelle von einem VIA.
3) Durch Setzen des Unterbrechungs-Sperr-Flags im Stapel während der Service-Routine. Das folgende Programm würde diese Aufgabe ausführen (erinnern Sie sich daran, daß das Unterbrechungs-Sperr-Flag das Bit 2 des Status-Registers ist und daß das Status-Register die oberste Eingabe in den Stapel darstellt (siehe Bild 12-1):

```
PLA                 ;HOLE STATUS-REGISTER
ORA   #%00000010    ;SETZE UNTERBRECHUNGS-SPERR-FLAG
PHA                 ;BRINGE STATUS-REGISTER VOM STAPEL
                    ;  ZURÜCK
```

RTI wird dann eine Rückkehr zum Hauptprogramm bewirken, wobei das gesamte Unterbrechungs-System gesperrt ist.

Beachten Sie jedoch, daß Sie sehr sorgfältig vorgehen müssen und die Unterbrechungen nicht automatisch wieder freigeben, da das Hauptprogramm überhaupt nicht wissen kann, daß Unterbrechungen nicht länger erlaubt sind. **Im allgemeinen sollten alle Unterbrechungs-Service-Routinen die Unterbrechungen wieder vor der Rückkehr freigeben. Jede andere Politik bedeutet, daß die Service-Routinen nicht transparent für das Hauptprogramm sind.**

Eine Fernschreiber-Unterbrechung

Zweck: Der Computer wartet auf zu empfangende Daten von einem Fernschreiber und speichert die Daten in den Speicherplatz 0040.

ACIA-UNTERBRECHUNGS-ROUTINE

a) Verwendung eines ACIA 6850

(7-Bit-Zeichen mit ungerader Parität und zwei Stop-Bits.)

Quellprogramm:

Hauptprogramm:

```
        LDX   #$FF         ;BRINGE STAPEL AN DAS ENDE VON
                           ;  SEITE 1
        TXS
        LDA   #%00000011   ;HAUPT-RESET DES ACIA
        STA   ACIACR
        LDA   #%11000101   ;GIB ACIA-UNTERBRECHUNG FREI
        STA   ACIACR
        CLI                ;GIB CPU-UNTERBRECHUNG FREI
HERE    HMP   HERE         ;"DUMMY"-HAUPTPROGRAMM
```

Unterbrechungs-Service-Routine:

```
        *=INTRP
        PHA                ;BEWAHRE AKKUMULATOR IN STAPEL AUF
        LDA   ACIADR       ;HOLE DATEN VON ACIA
        STA   $40          ;BEWAHRE DATEN AUF
        PLA                ;SPEICHERE AKKUMULATOR VOM STAPEL
                           ;  ZURÜCK
        RTI
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)		Befehl (Mnemonik)	
Hauptprogramm:				
0000	A2		LDX	#$FF
0001	FF			
0002	9A		TXS	
0003	A9		LDA	#%00000011
0004	03			
0005	8D		STA	ACIACR
0006, 0007	ACIACR			
0008	A9		LDA	#%11000101
0009	C5			
000A	8D		STA	ACIACR
000B, 000C	ACIACR			
000D	58		CLI	
000E	4C	HERE	JMP	HERE
000F	0E			
0010	00			
Unterbrechungs-Service-Routine:				
INTRP	48		PHA	
INTRP+1	AD		LDA	ACIADR
INTRP+2, INTRP+3	ACIADR			
INTRP+4	85		STA	$40
INTRP+5	40			
INTRP+6	68		PLA	
INTRP+7	40		RTI	

Erinnern Sie sich daran, daß der ACIA keinen RESET-Eingang besitzt, so daß ein "MASTER-RESET" (der die Steuerregister-Bits 0 und 1 beide "1" macht) erforderlich ist, bevor der ACIA verwendet wird. Die Konfiguration der ACIA-Steuerregister ist:

Bit 7 = 1, zur Freigabe der Empfangs-Unterbrechung
Bit 6 = 1, Bit 5 = 0 zur Sperrung der Sender-Unterbrechung und um RTS High (inaktiv) zu machen.
Bit 4 = 0, Bit 3 = 0, Bit 2 = 1, um 7-Bit-Daten mit gerader Parität und zwei Stop-Bits auszuwählen.
Bit 1 = 0, Bit 0 = 1 für ÷ 16 Takt (1760 Hz).

Um zu bestimmen, ob ein spezieller ACIA die Quelle einer Unterbrechung ist, muß die CPU das Unterbrechungs-Anforderungs-Bit, Bit 7 des Statusregisters, prüfen. Das Programm muß das Bit "Receive Data Register Full" (Statusregister-Bit 0) und das Bit "Transmitter Data Register Empty" (Statusregister-Bit 1) prüfen, um zwischen Empfangs- und Sende-Unterbrechungen zu unterscheiden.

Entweder löscht das Lesen des Empfangsdaten-Registers oder das Schreiben in das Sendedaten-Register das ACIA-Unterbrechungs-Anforderungsbit.

START-BIT-UNTERBRECHUNG

b) Verwendung eines VIA 6522
(Empfangene Daten werden sowohl zum Datenbit 7 wie zur Steuerleitung 1 des VIA geführt.)

Quellprogramm:

Hauptprogramm:

```
         LDX   #$FF          ;LEGE STAPEL AN DAS ENDE VON SEITE 1
         TXS
         LDA   #0
         STA   VIAPCR        ;MACHE ALLE STEUERLEITUNGEN
                             ; ZU EINGÄNGEN
         STA   VIADDRA
         LDA   #%10000010
         STA   VIAIFR        ;LÖSCHE START-BIT-UNTERBRECHUNGS-
                             ; FLAG
         STA   VIAIER        ;GIB STARTBIT-UNTERBRECHUNG VON VIA
                             ; FREI
         CLI                 ;GIB CPU-UNTERBRECHUNG FREI
HERE     JUMP  HERE          ;"DUMMY"-HAUPTPROGRAMM
```

Unterbrechungs-Service-Routine:

```
         *=INTRP
         PHA                 ;BEWAHRE AKKUMULATOR IN STAPEL AUF
         LDA   #%00000010
         STA   VIAIFR        ;LÖSCHE STARTBIT-UNTERBRECHUNGS-
                             ; FLAG
         STA   VIAIER        ;GIB STARTBIT-UNTERBRECHUNG FREI
         JSR   TTYRCV        ;HOLE DATEN VON TTY
         LDA   #%10000010
         STA   VIAIER        ;GIB STARTBIT-UNTERBRECHUNG WIEDER
                             ; FREI
         PLA                 ;SPEICHERE AKKUMULATOR VOM STAPEL
                             ; ZURÜCK
         RTI
```

Objektprogramm:

Speicher-Adresse (Hex)	Speicher-Inhalt (Hex)	Befehl (Mnemonik)		
Hauptprogramm:				
0000	A2		LDX	#$FF
0001	FF			
0002	9A		TXS	
0003	A9		LDA	#0
0004	00			
0005	8D		STA	VIAPCR
0006 0007	VIAPCR			
0008	8D		STA	VIADDRA
0009 000A	VIADDRA			
000B	A9		LDA	#%10000010
000C	82			
000D	8D		STA	VIAIFR
000E 000F	VIAIFR			
0010	8D		STA	VIAIER
0011 0012	VIAIER			
0013	58		CLI	
0014	4C	HERE	JMP	HERE
0015	14			
0016	00			
Unterbrechungs-Service-Routine:				
INTRP	48		PHA	
INTRP+1	A9		LDA	#%00000010
INTRP+2	02			
INTRP+3	8D		STA	VIAIFR
INTRP+4 INTRP+5	VIAIFR			
INTRP+6	8D		STA	VIAIER
INTRP+7 INTRP+8	VIAIER			
INTRP+9	20		JSR	TTYRCV
INTRP+10 INTRP+11	TTYRCV			
INTRP+12	A9		LDA	#%10000010
INTRP+13	82			
INTRP+14	8D		STA	VIAIER
INTRP+15 INTRP+16	VIAIER			
INTRP+17	68		PLA	
INTRP+18	40		RTI	

Das Unterprogramm TTYRCV ist die Fernschreiber-Empfangs-Routine, die im vorhergehenden Kapitel gezeigt wurde.

Die verwendete Flanke, die die Unterbrechung bewirkt, ist hier sehr wichtig. Der Übergang vom normalen "1"-Zustand (MARK) zum "0"-Zustand (SPACE) muß die Unterbrechung bewirken, da dieser Übergang den Start der Übertragung kennzeichnet. Es wird kein "0"-auf-"1"-Übergang auftreten, bis ein Nicht-Null-Datenbit empfangen wird.

Die Service-Routine muß die VIA-Unterbrechung sperren, da sonst jeder "1"-auf-"0"-Übergang im Zeichen eine Unterbrechung bewirken wird. Natürlich müssen Sie die VIA-Unterbrechung wieder freigeben, nachdem das ganze Zeichen gelesen worden ist.

Beachten Sie, wie VIA-Unterbrechungen freigegeben oder gesperrt werden. Bit 7 des Unterbrechungs-Freigaberegisters ist ein "Setz-Lösch-Steuer"-Bit. Wenn dieses Bit 0 ist, löschen darauf folgende 1-Bits die Unterbrechungsfreigabe-Bits und sperren daher die entsprechenden Unterbrechungen. Wenn dieses Bit 1 ist, setzen darauffolgende 1-Bits die Unterbrechungsfreigabe-Bits und geben daher die entsprechenden Unterbrechungen frei. Der Prozessor kann in Wirklichkeit nicht in das Bit 7 des Unterbrechungsflag-Registers schreiben, so daß entweder ein Freigabe- oder Sperr-Muster zum Löschen der Unterbrechungsflags verwendet werden kann. Erinnern sie sich an die Beschreibung des Unterbrechungsfreigabe-Registers und der Unterbrechungsflag-Register in den Bildern 12-2 und 12-3.

ALLGEMEINERE SERVICE-ROUTINEN

AUFGABEN FÜR ALLGEMEINE SERVICE-ROUTINEN

Allgemeinere Service-Routinen, die ein Teil eines vollständigen unterbrechungs-gesteuerten Systems sind, müssen folgende Aufgaben erfüllen:

1) **Aufbewahren aller Register, die in der Unterbrechungs-Service-Routine im Stapel verwendet werden, so daß das unterbrochene Programm korrekt wiederaufgenommen werden kann.**

Erinnnern Sie sich daran, daß der 6502 nur Push-Befehle für den Akkumulator und für das Statusregister P besitzt. Das "Pushen" des Statusregisters ist nach einer Unterbrechung nicht erforderlich, da die Unterbrechungs-Reaktion dies automatisch ausführt. Eine Routine zum Aufbewahren aller Register in den Stapel würde sein (wie früher gezeigt):

```
PHA         ;BEWAHRE AKKUMULATOR IN STAPEL
            AUF
TXA         ;BEWAHRE INDEXREGISTER X IM STAPEL
            ; AUF
PHA
TYA         ;BEWAHRE INDEXREGISTER Y IM STAPEL
            ; AUF
PHA
```

In einigen 6502-Programmen werden bestimmte Speicherplätze auf der Nullseite als zusätzliche Register behandelt. Derartige Speicherplätze müssen möglicherweise aufbewahrt und während Unterbrechungs-Service-Routinen wieder zurückgespeichert werden. Das Verfahren zum Aufbewahren des Inhalts des Speicherplatzes 0040 würde zum Beispiel sein:

```
LDA   $40   ;BEWAHRE SPEICHERPLATZ 0040 IM
            ; STAPEL AUF
PHA
```

Natürlich müssen nur jene Register, die von der Unterbrechungs-Service-Routine verwendet werden, aufbewahrt werden.

2) **Zurückspeichern aller Register vom Stapel nach Abschluß der Unterbrechungs-Service-Routine.** Erinnern Sie sich daran, daß Register in der entgegengesetzten Reihenfolge zurückgespeichert werden, in der sie aufbewahrt wurden.

3) **Ordnungsgemäße Freigabe und Sperren von Unterbrechungen.** Erinnern Sie sich daran, daß die CPU automatisch ihre Unterbrechungen sperrt, sobald sie eine hiervon empfängt.

Die Service-Routinen sollten transparent sein, soweit es das Unterbrechungsprogramm betrifft (d.h., es sollte keine zufälligen Effekte bewirken).

Jedes Standard-Unterprogramm, das von einer Unterbrechungs-Service-Routine verwendet wird, muß "re-entrant" sein, das heißt, es muß in dieses wieder eingetreten werden können. Wenn einige Unterprogramme nicht "re-entrant" gemacht werden können, muß die Unterbrechungs-Service-Routine verschiedene Versionen für ihre Verwendung besitzen.[7]

AUFGABEN

1) Eine Test-Unterbrechung

Zweck: Der Computer wartet, bis eine VIA-Unterbrechung auftritt, dann führt er den endlosen Schleifenbefehl

```
HERE    JMP    HERE
```

aus, bis die nächste Unterbrechung auftritt.

2) Eine Tastatur-Unterbrechung

Zweck: Der Computer wartet auf eine vierstellige Eingabe von einer Tastatur und plaziert die Stellen in die Speicherplätze 0040 bis 0043 (die erste empfangene Stelle in 0040). Jede Eingabe einer Stelle bewirkt eine Unterbrechung. Die vierte Eingabe sollte ein Sperren der Tastatur-Unterbrechung ergeben.

Beispiel:

Tastatur-Daten = 04, 06, 01, 07

Ergebnis: (0040) = 04
(0041) = 06
(0042) = 01
(0043) = 07

3) Eine Drucker-Unterbrechung

Zweck: Der Computer sendet vier Zeichen von den Speicherplätzen 0040 bis 0043 (beginnend mit 0040) zum Drucker. Jedes Zeichen wird durch eine Unterbrechung angefordert. Die vierte Übertragung sperrt auch die Drucker-Unterbrechung.

4) Eine Echtzeit-Takt-Unterbrechung

Zweck: Der Computer löscht anfangs den Speicherplatz 0040 und komplementiert dann den Speicherplatz 0040 jedesmal, wenn eine Echtzeittakt-Unterbrechung auftritt.

Wie würden sie das Programm ändern, so daß es den Speicherplatz 0040 nach jeweils zehn Unterbrechungen komplementiert? Wie würden sie das Programm ändern, so daß es den Speicherplatz 0040 während zehn Taktperioden auf null läßt, FF_{16} für fünf Taktperioden und so weiter kontinuierlich? Sie könnten die Verwendung einer Anzeige wünschen anstatt des Speicherplatzes 0040, so daß es leichter zu sehen ist.

5) Eine Fernschreiber-Unterbrechung

Zweck: Der Computer empfängt Daten von einem unterbrechenden ACIA 6850 und speichert die Zeichen in einen Puffer, beginnend im Speicherplatz 0040. Der Vorgang wird so lange fortgesetzt, bis der Computer einen "Wagenrücklauf" ($0D_{16}$) empfängt. Nehmen Sie an, daß die Zeichen 7-Bit-ASCII mit ungerader Parität sind. Wie würden Sie Ihr Programm zur Verwendung eines VIA ändern? Nehmen Sie an, daß das Unterprogramm TTYRCV verfügbar ist wie in dem vorhergehenden Beispiel. Schließen Sie den "Wagenrücklauf" als Abschluß-Zeichen in den Puffer mit ein.

LITERATUR

1. A. Osborne, "Einführung in die Microcomputer-Technik". TEWI-Verlag, München.

2. R. L. Baldridge, "Interrupts Add Power, Complexity to Microcomputer System Design", EDN, August 5, 1977, pp. 67-73.

3. L. Leventhal, "6800 Programmieren in Assembler" 1980, TEWI-Verlag, München.

4. MCS6500 Microcomputer Family Hardware Manual, MOS Technology Inc., pp. 104-108.

5. J. Gieryic, "SYM-1 6522-Based Timer", Micro, April 1979, pp. 11-31 to 11-32

6. M. L. DeJong, "A Simple 24-Hour Clock for the AIM 65", Micro, March 1979, pp. 10-5 to 10-7.

7. Frühere Besprechungen und einige praktische Beispiele der Entwicklung der 6502- und 6800-basierenden Systeme mit Unterbrechungen sehen im folgenden:

6502

T. Travis, "Patching a Program into a ROM", Electronics, September 1, 1976, pp. 98-101.

G. L. Zick and T. T. Sheffer, "Remote Failure Analysis of Micro-Based Instrumentation", Computer, September 1977, pp. 30-35.

6800

S. C. Baunach, "An Example of an M6800-based GPIB Interface", EDN, September 20, 1977, pp. 125-128.

L. E. Cannon and P. S. Kreager, "Using a Microprocessor: a Real-Life Application, Part 2 – Software", Computer Design, October 1975, pp. 81-89.

D. Fullager et al., "Interfacing Data Converters and Microprocessors", Electronics, December 8, 1976, pp. 81-89.

W. S. Holderby, "Designing a Microprocessor-based Terminal for Factory Data Collection", Computer Design, March 1977, pp. 81-88.

A. Lange, "OPTACON Interface Permits the Blind to 'Read' Digital Instruments", EDN, February 5, 1976, pp. 84-86.

J. D. Logan and P. S. Kreager, "Using a Microprocessor: a Real-Life Application, Part 1 – Hardware", Computer Design, September 1975, pp. 69-77.

A. Moore and M. Eidson, "Printer Control", Application Note available from Motorola Semiconductor Products, Phoenix, AZ.

M. C. Mulder and P. P. Fasang, "A Microprocessor Controlled Substation Alarm Logger", IECI '78 Proceedings – Industrial Applications of Microprocessors, March 20-22, 1978, pp.36-44.

Die "Proceedings of the IEEE's Industrial Electronics and Control Instrumentation Group's Annual Meeting" über "Industrial Applications of Microprocessors" enthalten zahlreiche interessante Artikel. Die entsprechenden Bände sind beim IEEE Service-Center, CP Department, 445 Hoes Lane, Piscataway, NJ 08854, erhältlich.

Kapitel 13
AUFGABEN-DEFINITION UND PROGRAMM-ENTWICKLUNG

DIE AUFGABEN DER SOFTWARE-ENTWICKLUNG

In den vorausgehenden Kapiteln haben wir uns auf das Schreiben kurzer Programme in Assemblersprache konzentriert. Wenn dies auch ein wesentlicher Punkt ist, so ist es nur ein kleiner Teil der gesamten Aufgabe der Software-Entwicklung. Obwohl das Schreiben von Assemblersprachen-Programmen für den Neuling eine Hauptaufgabe zu sein scheint, wird sie bald verhältnismäßig einfach. Sie sollten nunmehr mit den meisten der Standardmethoden für die Programmierung in Assemblersprache des Mikroprozessors 6502 vertraut sein. **Die nächsten vier Kapitel werden beschreiben, wie man Aufgaben in Form von Programmen formuliert und wie man kurze Programme zusammenfügt, um ein arbeitendes System zu bilden.**

STUFEN DER SOFTWARE-ENTWICKLUNG

Software-Entwicklung besteht aus mehreren Stufen. Bild 13-1 ist ein Flußdiagramm des Vorgangs der Software-Entwicklung.

Seine Stufen sind:

- Definition der Aufgabe
- Programm-Entwicklung
- Codierung
- Fehlersuche
- Testen
- Dokumentieren
- Wartung und Neu-Entwicklung

Jede dieser Stufen ist wichtig zum Aufbau eines funktionierenden Systems. Beachten Sie, daß die Codierung, das Schreiben von Programmen in einer Form, die der Computer versteht, nur eine von sieben Stufen darstellt.

RELATIVE BEDEUTUNG DER CODIERUNG

In der Tat ist die Codierung gewöhnlich die am einfachsten zu definierende und auszuführende Stufe. Die Regeln für das Schreiben von Computerprogrammen sind leicht zu erlernen. Sie variieren etwas von Computer zu Computer, jedoch die grundlegenden Techniken bleiben gleich. Wenige Software-Projekte geben Ärger infolge der Codierung. In der Tat ist die Codierung nicht der wichtigste Teil der Software-Entwicklung. Fachleute schätzen, daß ein Programmierer ein bis zehn voll fehlerfrei gemachte und dokumentierte Anweisungen pro Tag schreiben kann. Offensichtlich ist die bloße Codierung von ein bis zehn Anweisungen kaum eine Arbeit für einen ganzen Tag. Bei den meisten Software-Projekten benötigt die Codierung weniger als 25% der Zeit des Programmierers.

MESSEN DES FORTSCHRITTS IN DEN STUFEN

Die Messung des Fortschrittes in den anderen Stufen ist schwierig. Man kann sagen, daß die Hälfte des Programmes geschrieben wurde, aber Sie können kaum sagen, daß die Hälfte der Fehler entfernt wurde oder die Hälfte der Aufgabe definiert wurde. Zeit-Tabellen für derartige Stufen wie Programm-Entwicklung, Fehlersuche und Testen sind schwierig herzustellen. Viele Tage oder Wochen voller Anstrengung können unter Umständen in keinem offensichtlichen Fortschritt resultieren. Ferner kann eine unvollständige Arbeit in einer Stufe später in schwerwiegenden Problemen resultieren. Zum Beispiel kann eine schlechte Aufgaben-Definition oder Programm-Entwicklung die Fehlersuche und das Testen sehr schwierig machen. In einer Stufe gesparte Zeit kann sehr viel Zeit in späteren Stufen kosten.

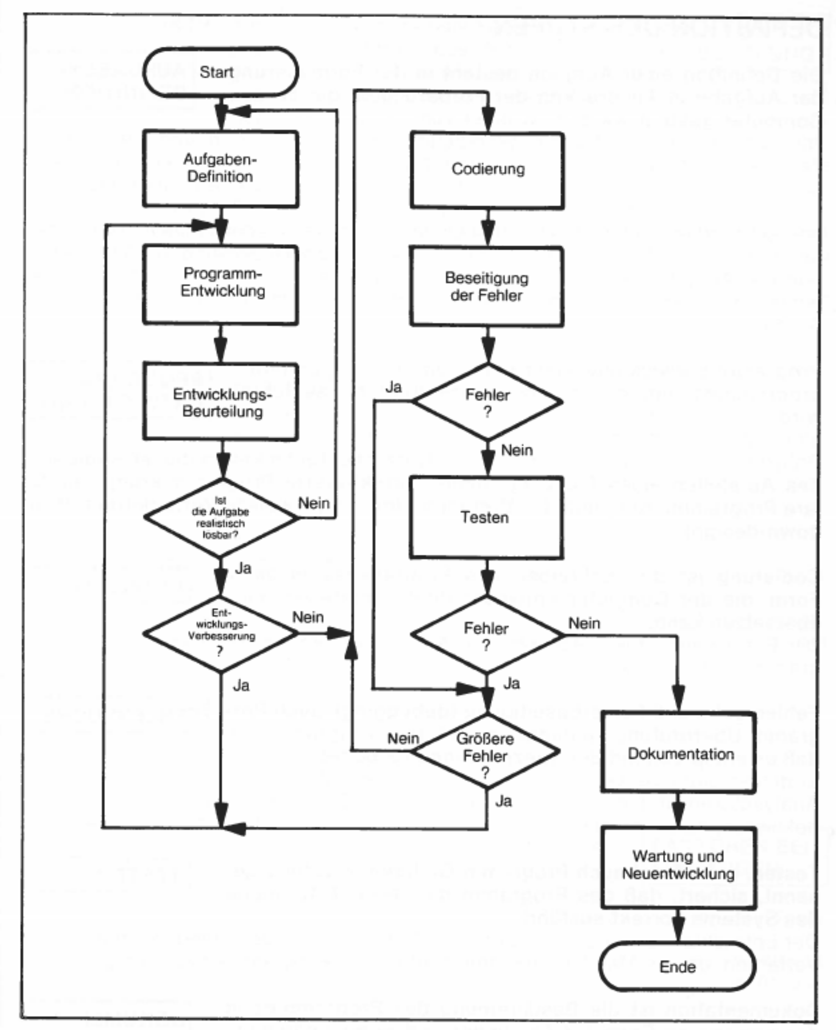


Bild 13-1. Flußdiagramm der Software-Entwicklung.

DEFINITION DER STUFEN

AUFGABEN-DEFINITION

Die Definition einer Aufgabe besteht in der Formulierung der Aufgabe in Ausdrücken der Forderungen, die an den Computer gestellt werden. Was ist zum Beispiel erforderlich, daß ein Computer ein Werkzeug steuert, eine Serie von elektrischen Tests ausführt, oder die Kommunikation zwischen einem zentralen Steuergerät und einem entfernt stehenden Instrument ausführt? Die Aufgaben-Definition erfordert, daß Sie die Form und die Geschwindigkeit der Eingaben und Ausgaben bestimmen, den erforderlichen Umfang und die Geschwindigkeit der Verarbeitung und die Arten der möglichen Fehler und deren Beseitigung. Die Definition der Aufgabe setzt eine einigermaßen definierte Vorstellung des Aufbaues eines computer-gesteuerten Systems voraus und definiert die Aufgaben und Anforderungen an den Computer.

PROGRAMM-ENTWICKLUNG

Programm-Entwicklung stellt einen Umriß des Computerprogrammes dar, das die definierte Aufgabe ausführen wird. In dieser Entwicklungsstufe werden die Aufgaben auf eine Weise beschrieben, in der sie leicht in ein Programm umgesetzt werden können. **Nützliche Techniken in dieser Stufe sind das Aufstellen eines Flußdiagrammes, strukturierte Programmierung, modulare Programmierung und das Verfahren der "schrittweisen Verfeinerung" (top-down-design).**

CODIERUNG

Codierung ist das Schreiben des Programmes in einer Form, die der Computer entweder direkt verstehen oder übersetzen kann.
Die Form kann Maschinensprache, Assemblersprache oder eine höhere Programmiersprache sein.

FEHLERSUCHE

Fehlersuche und Fehlerbeseitigung (debugging), auch Programm-Überprüfung genannt, macht das Programm so, daß es entsprechend der Spezifikationen arbeitet.
In dieser Stufe verwenden Sie Hilfsmittel wie Haltepunkte, Simulatoren, Logik-Analysatoren und In-Circuit-Emulatoren. Das Ende dieser Stufe ist schwer zu definieren, da sie niemals wissen, ob sie den letzten Fehler gefunden haben.

TESTEN

Testen, manchmal auch Programm-Gültigkeitsprüfung genannt, sichert, daß das Programm die Gesamt-Aufgaben des Systems korrekt ausführt.
Der Entwickler verwendet Simulatoren, Prüfgeräte und verschiedene statistische Verfahren, um ein Maß für die Eigenschaften des Programmes zu erhalten.

DOKUMENTATION

Dokumentation ist die Beschreibung des Programmes in der geeigneten Form für Anwender und Servicepersonal. Dokumentation gestattet auch dem Entwickler eine Programm-Bibliothek aufzustellen, so daß später folgende Aufgaben wesentlich einfacher sein werden. Flußdiagramme, Kommentare, Speicherpläne und eine Bibliothek bilden einige der Hilfsmittel, die bei der Dokumentation verwendet werden.

BETREUUNG UND NEU-ENTWICKLUNG

Betreuung und Neu-Entwicklung stellen die Wartung, Verbesserung und Erweiterung des Programmes dar. Natürlich muß der Entwickler bereit sein, Anwender-Probleme mit einem auf Computer basierenden Gerät zu handhaben. Es können hierzu Diagnostik-Verfahren oder Programme und andere Hilfsmittel erforderlich sein. Verbesserung oder Erweiterung des Programmes kann notwendig sein um eine neue Anforderung zu erfüllen oder neue Aufgaben zu erledigen.

Der Rest dieses Kapitels wird sich nur mit der Aufgaben-Definition und den Programm-Entwichlungsstufen befassen. Kapitel 14 wird die Fehlersuche und das Testen besprechen, und Kapitel 15 wird die Dokumentation, Erweiterung und Neu-Entwicklung behandeln. Wir werden alle Stufen in einigen einfachen System-Beispielen in Kapitel 16 zusammenfügen.

AUFGABEN-DEFINITION

Typische Mikroprozessor-Aufgaben erfordern eine gründliche Definition. Zum Beispiel, was muß ein Programm ausführen, um eine Waage, eine Registrierkasse oder einen Signal-Generator zu steuern? Offensichtlich bereitet es nicht wenig Mühe, allein die entsprechenden Aufgaben exakt zu definieren.

DEFINITION DER EINGABEN

Wie sollen wir die Definition beginnen? Es ist naheliegend mit den Eingaben anzufangen. **Hierzu sollten wir eine List aller Eingangs-Signale aufstellen, die der Computer in dieser Anwendung empfängt.**

Beispiele von Eingaben sind:

- Datenblöcke von Übertragungsleitungen
- Statusworte von peripheren Geräten
- Daten von A/D-Konvertern

FAKTOREN BEI EINGÄNGEN

Dann können wir folgende Fragen über jedes Eingangs-Signal aufstellen:

1) In welcher Form liegt es vor, das heißt, welche Signale wird der Computer tatsächlich empfangen?
2) Wann ist das Eingangssignal verfügbar und wie weiß der Prozessor, daß es verfügbar ist? Muß der Prozessor das Eingangssignal mit einem Abtast-Signal anfordern? Besitzen die Eingangssignale ihren eigenen Takt?
3) Wie lange sind die Eingangssignale verfügbar?
4) Wie oft ändern sie sich und wie weiß der Prozessor, daß sie sich geändert haben?
5) Bestehen die Eingangssignale aus einer Sequenz oder einem Datenblock? Ist die Reihenfolge wichtig?
6) Was sollte geschehen, wenn die Daten Fehler enthalten? Diese können Übertragungsfehler enthalten, falsche Daten, falsche Reihenfolge, zusätzliche Daten etc.
7) Besitzt das Eingangssignal eine Beziehung zu anderen Eingangs- oder Ausgangs-Signalen?

DEFINITION DER AUSGANGS-SIGNALE

Der nächste Schritt besteht in der Definition der Ausgaben. **Wir müssen alle Ausgangssignale auflisten, die der Computer erzeugen muß.** Beispiele von Ausgangssignalen beinhalten:

- Datenblöcke zu Übertragungsleitungen
- Steuerworte zu peripheren Bausteinen
- Daten zu D/A-Wandlern

Dann können wir folgende Fragen über jedes Ausgangssignal stellen:

1) In welcher Form liegen sie vor, das heißt, welche Signale muß der Computer erzeugen?
2) Wann müssen sie verfügbar sein, und wie weiß der periphere Baustein, daß sie verfügbar sind?
3) Wie lange müssen sie verfügbar sein?
4) Wie oft müssen sie geändert werden und wie weiß der periphere Baustein, daß sie sich geändert haben?
5) Liegen die Ausgangssignale in einer bestimmten Reihenfolge vor? Ist die Reihenfolge wichtig?
6) Was sollte geschehen, um Übertragungsfehler zu vermeiden oder Fehler von peripheren Bausteinen zu erkennen und zu berichtigen?
7) Besteht ein Zusammenhang mit anderen Eingangs- und Ausgangs-Signalen?

VERARBEITUNGS-ABSCHNITT

Zwischen dem Lesen von Eingangsdaten und dem Senden von Ausgangs-Resultaten liegt der Verarbeitungs-Abschnitt. **Hier müssen wir exakt bestimmen, wie der Computer die Eingangsdaten verarbeiten muß. Die Fragen sind:**

FAKTOREN DER VERARBEITUNG

1) Wie lautet das grundlegende Verfahren (Algorithmus) zur Umwandlung von Eingangsdaten in Ausgangs-Ergebnisse?
2) Welche zeitlichen Beschränkungen liegen vor? Diese können Datengeschwindigkeit, Verzögerungszeiten, die Zeitkonstanten von Eingabe- und Ausgabe-Bausteinen enthalten etc.
3) Welche Speicher-Beschränkungen liegen vor? Bestehen Grenzen bei der Größe des Programmspeichers oder Datenspeichers, oder bei der Größe des Puffers?
4) Welche Standardprogramme oder Tabellen müssen verwendet werden? Welche Anforderungen bestehen an diese?
5) Welche speziellen Fälle existieren und wie sollte das Programm diese verarbeiten?
6) Wie genau müssen die Ergebnisse sein?
7) Wie sollte das Programm auf Verarbeitungsfehler oder spezielle Bedingungen, wie Überlauf, Unterlauf u.ä. reagieren?

VERHALTEN BEI FEHLERN

Ein wichtiger Faktor in zahlreichen Anwendungen ist das Verhalten bei Fehlern. Natürlich muß der Entwickler Vorsorge treffen, um allgemeine Fehler aufzudecken und Fehlfunktionen zu erkennen. **Hierzu sind vom Entwickler folgende Fragen zu stellen:**

FEHLER-BETRACHTUNGEN

1) Welche Fehler können auftreten?
2) Welche Fehler sind am wahrscheinlichsten? Wenn das System von einer Person bedient wird, sind menschliche Fehler am häufigsten. Nach den menschlichen Fehlern treten am ehesten Übertragungsfehler auf, darauf folgen mechanische, elektrische, mathematische oder Verarbeitungsfehler.
3) Welche Fehler sind im System nicht sofort offensichtlich? Ein spezielles Problem ist das Auftreten von Fehlern, die vom System oder Bedienenden nicht als solche sofort erkennbar sind.
4) Wie kann sich das System von Fehlern mit einem minimalen Verlust von Zeit und Daten wieder erholen, und erkennen, daß ein Fehler aufgetreten ist?
5) Welche Fehler oder Fehlfunktionen verursachen das gleiche System-Verhalten? Wie können diese Fehler oder Fehlfunktionen für Diagnostikzwecke unterschieden werden?
6) Welche Fehler betreffen spezielle Systemverfahren? Zum Beispiel, erfordern Paritätsfehler ein Zurücksenden der Daten?

Eine andere Frage ist: Wie kann der Servicetechniker systematisch die Quelle einer Fehlfunktion finden, ohne ein entsprechender Spezialist zu sein? Hier können eingebaute Testprogramme, spezielle Diagnostik- oder Kennzeichen-Analyse helfen.

MENSCHLICHE FAKTOREN

WECHSELWIRKUNG MIT DEM BEDIENENDEN

Zahlreiche, auf Mikroprozessoren basierende Systeme beinhalten eine Wechselwirkung mit dem Menschen. **Während des gesamten Entwicklungs-Vorganges müssen menschliche Faktoren für derartige Systeme in Betracht gezogen werden. Unter den Fragen, die ein Entwickler stellen muß, sind:**

1) Welches Eingabe-Verfahren ist das natürlichste für einen Menschen?
2) Kann der Bedienende leicht feststellen, wie er die Eingabe-Operationen beginnen, fortsetzen und beenden muß?
3) Wie wird der Bedienende über Prozedur-Fehler und Geräte-Fehlfunktionen informiert?
4) Welche Fehler wird der Bedienende am wahrscheinlichsten machen?
5) Wie erfährt der Bedienende, daß er Daten falsch eingegeben hat?
6) Liegen die Anzeigen in einer Form vor, die der Bedienende leicht lesen und verstehen kann?
7) Ist die Reaktion des Systems für den Bedienenden geeignet?
8) Ist das System leicht zu bedienen?
9) Gibt es geeignete Richtlinien für einen unerfahrenen Bedienenden?
10) Gibt es abgekürzte Verfahren und zusätzliche Optionen für den erfahrenen Bedienenden?
11) Kann der Bedienende immer den Zustand eines Systems nach Unterbrechungen bestimmen oder zurücksetzen?

Ein System so aufzubauen, daß es von Menschen leicht bedient werden kann, ist schwierig. Der Mikroprozessor kann das System leistungsfähiger, flexibler und vielseitiger machen. Der Entwickler muß jedoch die menschlichen Gesichtspunkte berücksichtigen, die den Nutzen und die Attraktivität eines Systems gewaltig erhöhen können, sowie die Produktivität des Bedienenden.

BEISPIELE

Reaktion auf einen Schalter

DEFINITION DES SCHALTERS- UND LAMPENSYSTEMS

SCHALTER- UND LAMPEN-EINGANG

Bild 13-2 zeigt ein einfaches System, in dem das Eingangsignal von einem einfachen einpoligen Schalter (Schließer) kommt und das Ausgangssignal zu einer einzelnen LED-Anzeige führt. Als Reaktion auf ein Schließen des Schalters schaltet der Prozessor die Anzeige eine Sekunde lang ein.
Dieses System sollte leicht zu definieren sein.

Wir wollen zuerst das Eingangssignal und die Antwort auf jede vorher aufgeworfenen Fragen darstellen:

1) Das Eingangs-Signal stellt ein einzelnes Bit dar, das entweder "0" (Schalter geschlossen) oder "1" (Schalter offen) ist.
2) Das Eingangssignal ist immer verfügbar und braucht nicht angefordert zu werden.
3) Das Eingangssignal ist für wenigstens mehrere Millisekunden nach dem Schließen verfügbar.
4) Das Eingangsssignal wird sich selten mehr als einmal innerhalb einiger Sekunden ändern. Der Prozessor muß das Prellen des Schalters handhaben können. Der Prozessor muß den Schalter überwachen, um zu bestimmen, wann er geschlossen ist.
5) Es gibt keine bestimmte Reihenfolge der Eingangsignale.
6) Die offensichtlichen Eingangsfehler sind ein Defekt des Schalters, Fehler in der Eingangs-Schaltung und der Fall, daß der Bedienende versucht, den Schalter zu schließen, bevor ein ausreichendes Zeit-Intervall verstrichen ist. Wir werden die Handhabung dieser Fehler später besprechen.
7) Das Eingangssignal hängt nicht von anderen Eingangs- oder Ausgangssignalen ab.

SCHALTER UND LAMPEN-AUSGANGSSIGNALE

Der nächste Schritt in der Definition des Systems besteht im Untersuchen des Ausgangssignales. Die Antworten auf diese Fragen sind:

1) Das Ausgangssignal ist ein einziges Bit, das "0" zum Einschalten der Anzeige, "1" zum Ausschalten ist.
2) Es gibt keine zeitlichen Beschränkungen des Ausgangssignales. Der periphere Baustein braucht nicht über die Verfügbarkeit der Daten informiert werden.
3) Wenn die Anzeige eine LED ist, müssen die Daten nur für einige wenige Millisekunden mit einer Pulsfrequenz von etwa 100 mal pro Sekunde vorliegen. Der Betrachter wird eine kontinuierlich leuchtende Anzeige sehen.
4) Die Daten müssen sich nach einer Sekunde ändern (abschalten).
5) Es gibt keine Reihenfolge der Ausgangssignale.
6) Die möglichen Fehler des Ausgangssignales sind ein Defekt der Anzeige und ein Fehler in der Ausgangs-Schaltung.
7) Das Ausgangs-Signal hängt nur von dem Schalter-Eingangs-Signal und der Zeit ab.

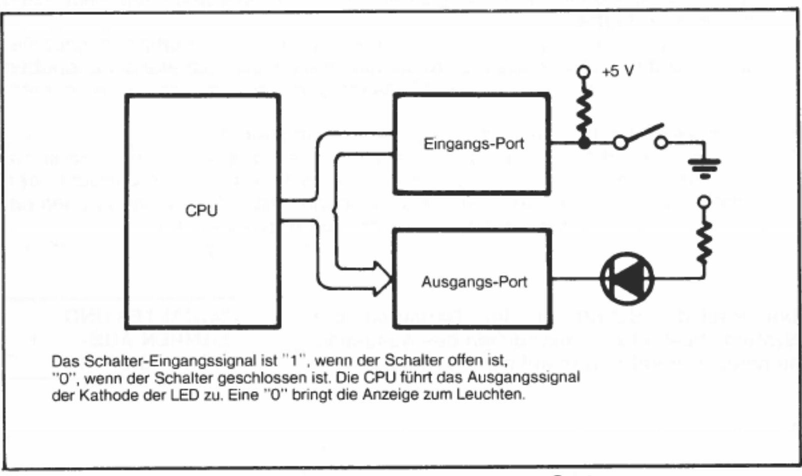
Der Verarbeitungs-Abschnitt ist extrem einfach. Sobald der Schalter-Eingang logisch "0" wird, schaltet die CPU die Lampe (eine logische "0") eine Sekunde lang ein. Es bestehen keinerlei zeitliche oder Speicher-Beschränkungen.

HANDHABUNG VON SCHALTER- UND LAMPENFEHLERN

Sehen wir uns nun die möglichen Fehler und Fehlfunktionen an. Diese sind:

1) Ein weiteres Schließen des Schalters, bevor eine Sekunde verstrichen ist.
2) Schalter-Fehler.
3) Anzeige-Fehler.
4) Computer-Fehler.

Sicher ist der erste Fehler der wahrscheinlichste. Die einfachste Lösung besteht für den Prozessor darin, ein Schließen des Schalters zu ignorieren, bis eine Sekunde verstrichen ist. Diese kurze zeitliche Periode wird sicher für den Bedienenden schwer erkennbar sein. Ferner bedeutet das Ignorieren des Schalters während dieser Periode, daß keine Entprell-Schaltung oder Software hierfür erforderlich ist, da das System in keiner Weise auf das Prellen reagiert.



Das Schalter-Eingangssignal ist "1", wenn der Schalter offen ist, "0", wenn der Schalter geschlossen ist. Die CPU führt das Ausgangssignal der Kathode der LED zu. Eine "0" bringt die Anzeige zum Leuchten.

Bild 13-2. Das Schalter- und Lampen-System.

Offensichtlich können die drei letzten Fehler nicht vorhersehbare Ergebnisse liefern. Die Anzeige kann auf Ein bleiben, auf Aus, oder den Zustand willkürlich ändern. Einige mögliche Wege zur Isolierung des Fehlers würden sein:

1) Lampentest-Hardware zur Prüfung der Anzeige, d.h. eine Taste, die die Lampe unabhängig vom Prozessor einschaltet.
2) Eine direkte Verbindung mit dem Schalter, um seine Arbeitsweise zu prüfen.
3) Ein Diagnostik-Programm, das die Eingangs- und Ausgangs-Schaltungen prüft.

Wenn sowohl die Anzeige wie der Schalter funktionieren, besitzt der Computer einen Fehler. Ein Servicetechniker mit entsprechenden Geräten kann die Ursache des Fehlers bestimmen.

Ein Speicherlader mit Schaltern

DEFINIEREN EINES SPEICHERLADERS MIT SCHALTERN

Bild 13-3 zeigt ein System, mit dem der Anwender Daten in jeden Speicherplatz in einem Mikrocomputer eingeben kann. Ein Eingangsport, DPORT, liest Daten von acht Kippschaltern.
Der andere Eingangsport, CPORT, wird zum Lesen von Steuer-Informationen verwendet. Es gibt drei Moment-Schalter: Hohe Adresse, niedrige Adresse und Daten.
Das Ausgangs-Signal ist der Wert der letzten vollständigen Eingabe von den Datenschaltern. Es werden acht LEDs für die Anzeige verwendet.

Das System wird natürlich außerdem verschiedene Widerstände, Puffer und Treiber benötigen.

Wir werden zuerst die Eingangssignale prüfen. Die Eigenschaften der Schalter sind die gleichen wie im vorhergehenden Beispiel. Es gibt jedoch eine bestimmte Reihenfolge der Eingangssignale:

1) Der Bedienende muß die Datenschalter entsprechend der acht höchsten Bits einer Adresse setzen, dann
2) den Schalter "Hohe-Adresse" drücken. Die hohen Adressen-Bits werden auf den Lampen aufscheinen und das Programm wird die Daten als das hohe Byte der Adresse interpretieren.
3) Dann muß der Bedienende die Datenschalter auf den Wert des niedrigstwertigen Bytes der Adresse stellen und
4) die Taste "Niedrige-Adresse" drücken. Die niedrigen Adressenbits werden auf den Lampen aufscheinen und das Programm wird die Daten als das niedrige Byte der Adresse ansehen.
5) Schließlich muß der Bedienende die gewünschten Daten auf den Datenschaltern einstellen und
6) den Schalter "Daten" drücken. Die Anzeige wird nun die Daten zeigen und das Programm speichert die Daten in den Speicher an den vorher eingegebenen Adressen.

Der Bedienende kann den Vorgang zur Eingabe eines ganzen Programmes wiederholen. Natürlich haben wir auch in dieser vereinfachten Situation zahlreiche mögliche Sequenzen zu betrachten. Wie können wir mit fehlerhaften Sequenzen fertig werden und das System leicht zu bedienen machen?

Die Ausgabe ist kein Problem. Nach jeder Eingabe sendet das Programm das Komplement der Eingangs-Bits zu den Anzeigen (da die Anzeigen aktiv low sind). Die Ausgangsdaten verbleiben unverändert, bis zur nächsten Eingabe-Operation.

Der Verarbeitungs-Abschnitt bleibt ziemlich einfach. Es gibt keine zeitlichen oder Speicher-Beschränkungen. Das Programm kann die Schalter entprellen, indem es einige wenige Millisekunden wartet und muß dann die komplementierten Daten zu den Anzeigen liefern.

Die wahrscheinlichsten Fehler sind Irrtümer des Bedienenden. Diese beinhalten:

HANDHABUNG DER FEHLER DES SPEICHER-LADERS

1) Falsche Eingaben
2) Falsche Reihenfolge
3) Unvollständige Eingaben, z.B. Vergessen von Daten.

Das System muß imstande sein, diese Probleme in einer vernünftigen Weise zu handhaben, da sie im tatsächlichen Betrieb mit Sicherheit auftreten.

Der Entwickler muß auch die Einflüsse von Fehlern des Gerätes beachten. So wie vorher sind die möglichen Schwierigkeiten:

1) Schalter-Fehler
2) Anzeige-Fehler
3) Computer-Fehler

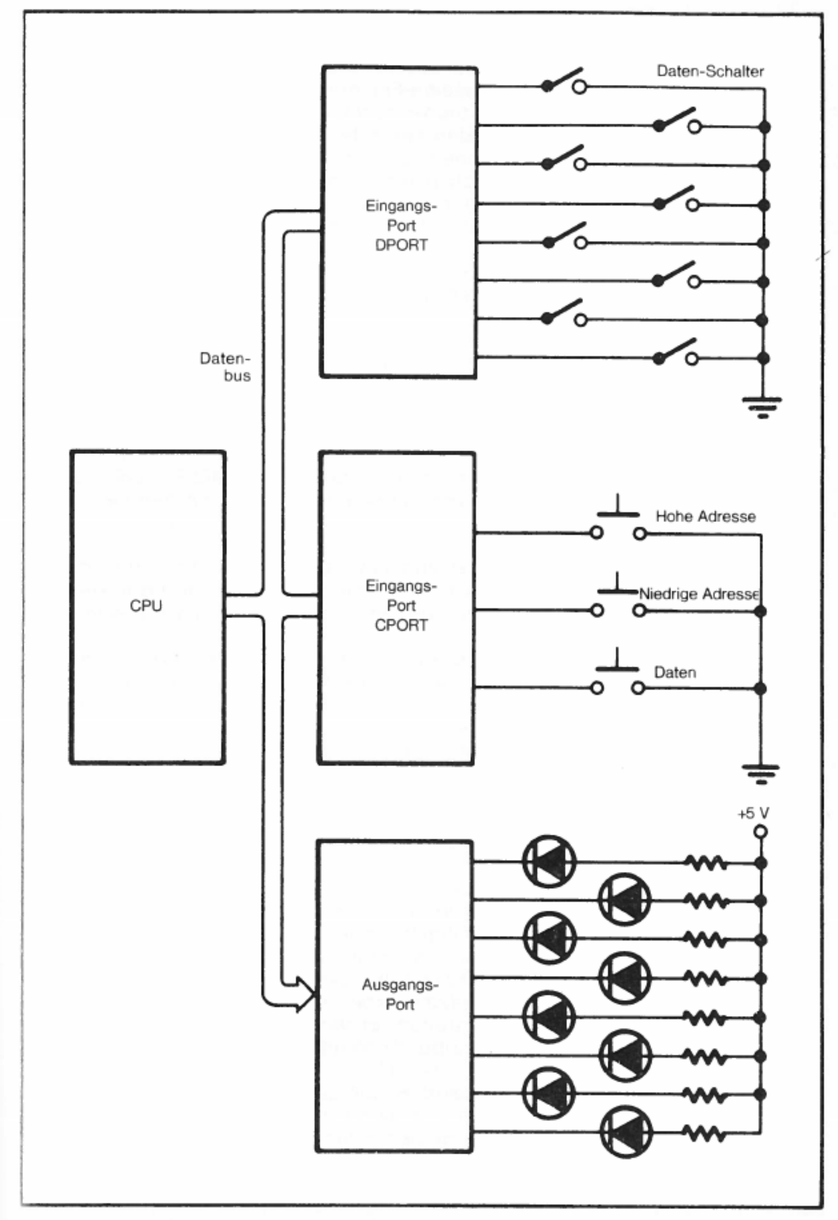


Bild 13-3. Der Speicher-Lader mit Schaltern.

In diesem System müssen wir jedoch mehr Aufmerksamkeit der Tatsache schenken, wie diese Fehler das System beeinflussen. Ein Computer-Fehler wird wahrscheinlich ein sehr ungewöhnliches Verhalten des Systems bewirken, und wird leicht festzustellen sein. Ein Anzeige-Fehler kann nicht unmittelbar erkennbar sein. Hier würde die Möglichkeit eines Lampentests dem Bedienenden gestatten, ihr Funktionieren zu überprüfen. Beachten Sie, daß wir jede LED separat testen wollen, um den Fall zu erkennen, bei dem Ausgangsleitungen gegeneinander kurzgeschlossen sind. Zusätzlich könnte der Bedienende nicht unmittelbar Schalter-Fehler feststellen. Jedoch sollte der Bedienende sie bald erkennen und durch ein Verfahren der Elimination feststellen, welcher Schalter defekt ist.

KORREKTUR DER FEHLER DES BEDIENENDEN BEIM SPEICHERLADER

Sehen wir uns einige der möglichen Fehler des Operators an. Typische Fehler würden sein:

1) Falsche Daten
2) Falsche Reihenfolge der Eingaben oder Schalter
3) Versuch, neue Eingaben durchzuführen, ohne vollständige Eingabe der Laufenden.

Der Bedienende wird wahrscheinlich Fehler bemerken, sobald sie auf den Anzeigen aufscheinen. Was wäre ein annehmbares Verfahren für den Bedienenden? Einige der Möglichkeiten sind:

1) Der Bedienende muß das Eingabe-Verfahren abschließen, d.h. die niedrigen Adressen und Daten eingeben, wenn der Fehler in den hohen Adressen auftritt. Natürlich ist dieses Verfahren mühsam und würde nur dazu dienen, den Bedienenden zu verärgern.
2) Der Bedienende muß den Eingabe-Prozess neu starten, indem er zu den Eingabe-Schritten für die hohe Adresse zurückkehrt. Diese Lösung ist nützlich, wenn der Fehler in den hohen Adressen auftrat, zwingt jedoch den Bedienenden frühere Daten wieder einzugeben, wenn sich der Fehler in den niedrigen Adressen- oder Datenstufen befunden hat.
3) Der Bedienende kann jeden Teil der Sequenz zu jeder Zeit eingeben, indem er einfach die Datenschalter auf die gewünschten Daten stellt und die entsprechende Taste drückt. Dieses Verfahren gestattet dem Bedienenden, Korrekturen an jedem Punkt der Sequenz auszuführen.

Diese Verfahren sollte immer jenem vorgezogen werden, das keine unmittelbare Fehlerkorrektur gestatten, aus einer Vielzahl von abschließenden Schritten besteht, oder Daten in das System eingibt, ohne dem Bedienenden einen Abschlußtest zu gestatten. Jede zusätzliche Komplikation in der Hardware oder Software wird durch eine erhöhte Effizienz des Bedienenden belohnt. Sie sollten immer den Mikrocomputer die mühsame Arbeit ausführen lassen und das Erkennen willkürlicher Sequenzen gestatten. Er wird niemals müde und vergißt niemals, was sich im Bedienungshandbuch befand.

Eine weitere nützliche Eigenschaft wären Status-Lampen, die die Bedeutung der Anzeige definiert. Drei Status-Lampen, markiert mit "Hohe Adresse", "Niedrige Adresse" und "Daten", würden dem Bedienenden wissen lassen, was er eingegeben hat und er sich nicht merken muß, welche Tasten er gedrückt hat. Der Prozessor müsste die Sequenz überwachen, jedoch die zusätzliche Komplikation in der Software würde die Aufgabe des Bedienenden sehr vereinfachen. Natürlich würden drei getrennte Sätze von Anzeigen plus der Möglichkeit des Prüfens eines Speicherplatzes noch nützlicher für den Bedienenden sein.

Wir sollten beachten, daß, obwohl wir die Wechselwirkung mit dem Bedienenden hervorgehoben haben, die Maschinen- oder System-Wechselwirkung zahlreiche gleiche Eigenschaften besitzt. Der Mikroprozessor sollte die Arbeit ausführen. Wenn durch Komplizierung der Aufgabe des Mikroprozessors das Aufdecken von Fehlern einfacher gemacht und die Ursachen der Fehler verdeutlicht werden, wird das ganze System besser arbeiten und leichter zu warten sein. Beachten Sie, daß Sie Überlegungen über die Verwendung des Systems und den Service nicht erst am Ende des Software-Entwicklungsprozesses machen sollten. Stattdessen sollten Sie diese bereits in der Definitions-Phase anstellen.

Ein Verifikations-Terminal

DEFINITION EINES VERIFIKATIONS-TERMINALS

Bild 13-4 ist ein Blockschaltbild eines einfachen Kreditkarten-Überprüfungs-Terminals. Ein Eingangsport leitet Daten von einer Tastatur her (siehe Bild 13-5), der andere Eingangsport nimmt Verifikations-Daten von einer Übertragungsleitung auf.
Ein Ausgangsport sendet Daten zu einem Satz von Anzeigen (siehe Bild 13-6). Ein anderer sendet die Kreditkarten-Nummer zum Zentralcomputer. Ein dritter Ausgangsport schaltet eine Lampe ein, wann immer das Terminal bereit zur Aufnahme einer Anfrage ist, eine andere Lampe, wenn der Bedienende die Information sendet. Die "BELEGT"-Lampe wird abgeschaltet, wenn die Antwort zurück kommt. Natürlich wird die Eingabe und die Ausgabe von Daten komplexer als im vorhergehenden Fall sein, obwohl die Verarbeitung noch ziemlich einfach ist.

Zusätzliche Anzeigen können sehr nützlich sein, um die Bedeutung der Antwort hervorzuheben. Zahlreiche Terminals verwenden ein grünes Licht für "Ja" ein rotes Licht für "Nein" und ein gelbes Licht für "beim Manager rückfragen". Beachten Sie, daß diese Lampen deutlich mit ihrer Bedeutung zu kennzeichnen sind, um auch mit der Situation fertig zu werden, bei der der Bedienende farbenblind ist.

EINGÄNGE ZUM VERIFIKATIONS-TERMINAL

Sehen wir uns zuerst das Tastatur-Eingangs-Signal an. Dieses ist natürlich verschieden von Schalter-Eingangs-Signalen, da die CPU irgendeine Möglichkeit besitzten muß, neue Daten zu unterscheiden. **Wir wollen annehmen, daß jedes Schließen einer Taste einen eindeutigen Hexadezimal-Code liefert (wir können alle Tasten in eine Ziffer codieren, da es zwölf Tasten gibt) und einen Abtast-Impuls. Das Programm wird den Abtast-Impuls zu erkennen haben und muß die Hexadezimalzahl holen, die die Taste identifiziert.** Es gibt hier eine zeitliche Einschränkung, da das Programm keine Daten oder Abtast-Impulse auslassen darf. Die Beschränkung ist nicht schwerwiegend, da Tastatur-Eingaben wenigstens einige Millisekunden voneinander entfernt sind.

Das Übertragungs-Eingangssignal besteht ähnlich aus einer Serie von Zeichen, von dene jedes durch einen Abtast-Impuls identifiziert wird (vielleicht von einem UART). Das Programm wird jeden Abtast-Impuls zu erkennen haben und das Zeichen holen müssen. Die Daten, die über die Übertragungsleitung gesendet werden, sind gewöhnlich in Nachrichten organisiert. Eine mögliche Nachrichten-Form ist:

1) Einführungszeichen, oder Nachrichten-Kopf (Header)
2) Bestimmungs-Adresse des Terminals.
3) Codiert es Ja oder Nein.
4) Abschlußzeichen (Trailer).

Das Terminal wird den Nachrichten-Kopf prüfen, die Bestimmungs-Adresse lesen und feststellen, ob die Nachricht für das Terminal vorgesehen ist. Wenn die Nachricht für das Terminal bestimmt ist, nimmt das Terminal die Daten an. Die Adresse könnte (und dies ist häufig der Fall) im Terminal fest verdrahtet sein, so daß das Terminal nur für dieses bestimmte Nachrichten empfängt. Diese Lösung vereinfacht die Software auf Kosten einiger Flexibilität.

AUSGANGSSIGNALE DES VERIFIKATIONS-TERMINALS

Das Ausgangssignal ist komplexer als in den früheren Beispielen. Wenn die Anzeigen gemultiplext werden, muß der Prozessor nicht nur die Daten zum Anzeige-Port senden, sondern muß auch die Daten zu einer bestimmten Anzeige leiten.
Wir werden entweder einen separaten Steuerport oder einen Zähler und Decoder hierfür benötigen. Beachten Sie, daß Hardware-Steuerungen führende Nullen so lange ausblenden können, wie die erste Stelle in einer mehrstelligen Zahl niemals null ist. Die Software kann diese Aufgabe ebenfalls übernehmen. Zeitliche Einschränkungen beinhalten die Impulslänge und Frequenz, die zur Erzeugung einer scheinbar kontinuierlichen Anzeige für den Bedienenden erforderlich ist.

Das Kommunikations-Ausgangs-Signal besteht aus einer Serie von Zeichen mit einem speziellen Format. Das Programm wird auch die zwischen den Zeichen erforderliche Zeit zu berücksichtigen haben. Ein mögliches Format für die Ausgangs-Nachricht ist:

1) Nachrichten-Kopf (Header)
2) Terminal-Adresse
3) Kreditkarten-Nummer
4) Abschluß-Zeichen (Trailer)

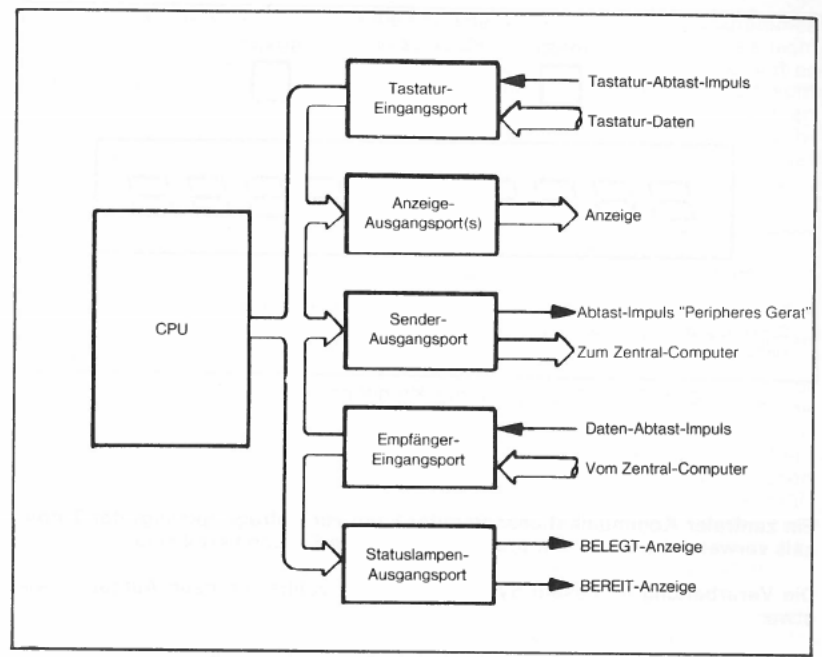


Bild 13-4. Blockschaltbild des Kreditüberprüfungs-Terminals.

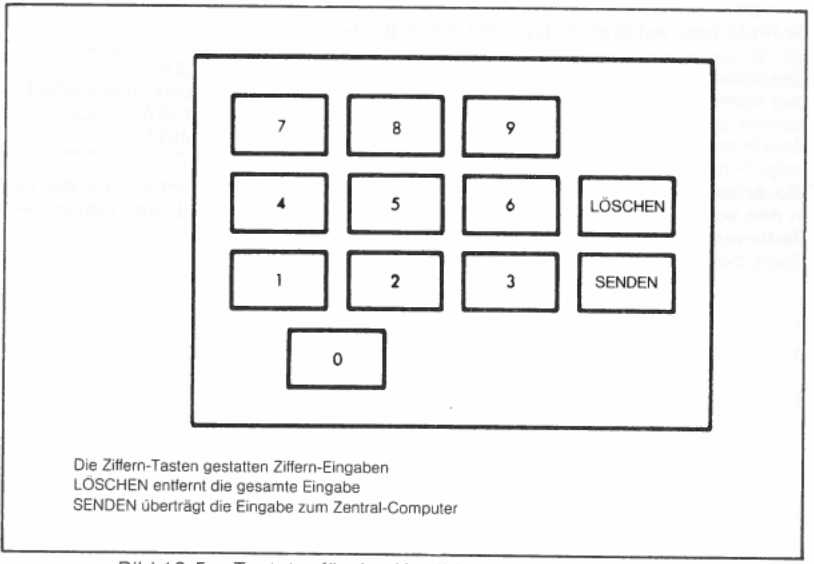
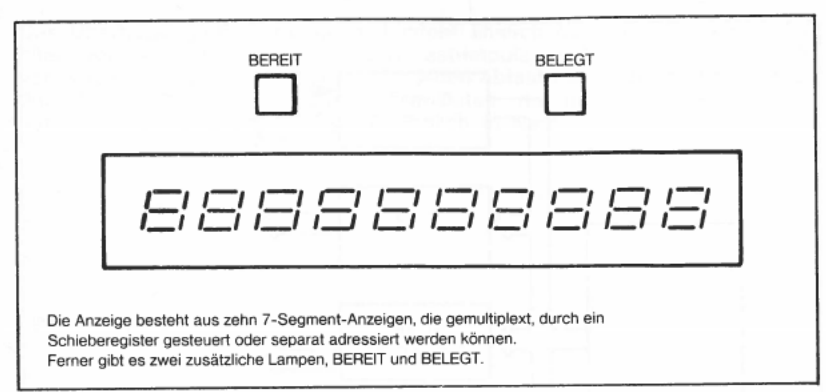


Bild 13-5. Tastatur für das Kreditüberprüfungs-Terminal



Die Anzeige besteht aus zehn 7-Segment-Anzeigen, die gemultiplext, durch ein Schieberegister gesteuert oder separat adressiert werden können.
Ferner gibt es zwei zusätzliche Lampen, BEREIT und BELEGT.

Bild 13-6. Anzeige für das Kreditüberprüfungs-Terminal.

Ein zentraler Kommunikationscomputer kann zur Abfrage (polling) der Terminals verwendet werden, der prüft, ob Daten zum Senden bereit sind.

Die Verarbeitung in diesem System beinhaltet zahlreiche neue Aufgaben wie etwa:

1) Identifizierung der Steuertasten durch eine Zahl und Ausführung der entsprechenden Tätigkeiten.
2) Hinzufügen des Kopf- und Abschlußzeichens für die abgehende Nachricht.
3) Erkennen des Kopf- und Abschlußzeichens in der zurückkehrenden Nachricht.
4) Prüfen der ankommenden Terminal-Adresse.

FEHLER-HANDHABUNG BEIM VERIFIKATIONS-TERMINAL

Beachten Sie, daß keine der Aufgaben irgendwelche komplexe Arithmetik beinhaltet oder irgendwelche schwerwiegende zeitliche oder Speicher-Beschränkungen.

Die Anzahl der möglichen Fehler in diesem System ist natürlich viel größer als in den vorhergehenden Beispielen. Wir wollen zuerst die möglichen Fehler des Bedienenden betrachten.
Diese beinhalten:

1) Falsche Eingabe der Kreditkarten-Nummer.
2) Versuch des Sendens einer unvollständigen Kreditkarten-Nummer.
3) Versuch des Sendens einer weiteren Zahl, während der Zentralcomputer bereits eine verarbeitet.
4) Löschen nicht existierender Eingaben.

Einigen dieser Fehler kann durch eine korrekte Strukturierung des Programmes leicht begegnet werden. Zum Beispiel sollte das Programm die Sendetaste nicht annehmen, bis die Kreditkarten-Nummer vollständig eigegeben wurde und es sollte jegliche zusätzliche Tastatur-Eingaben ignorieren, bis die Antwort vom Zentralcomputer zurückkommt. Beachten Sie, daß der Bedienende erkennen will, daß die Eingabe nicht gesendet wurde, da die Besetzt-Lampe nicht leuchtet. Der Bedienende wird auch wissen wollen, wann die Tastatur abgeschaltet war (wenn das Programm die Tastatur-Eingaben ignoriert), da die Eingaben nicht auf der Anzeige aufscheinen und die Bereit-Lampe aus sein wird.

KORRIGIEREN VON TASTATUR-FEHLERN

Falsche Eingaben sind ein offensichtliches Problem. Wenn der Bedienende einen Fehler erkannt hat, kann er oder sie die Löschtaste verwenden, um Korrekturen auszuführen.
Der Bedienende würde es wahrscheinlich bequemer finden, zwei Löschtasten zu haben, von denen eine die letzte Eingabe löscht und eine andere die gesamte Eingabe. Dies würde sowohl die Situation, in der der Bedienende den Fehler unmittelbar erkennt beherrschen, sowie die Situation, in der der Bedienende den Fehler relativ spät erkennt. Der Bedienende sollte imstande sein, Fehler unmittelbar zu korrigieren und möglichst wenige Tasten zu wiederholen haben. Der Bedienende wird jedoch eine bestimmte Anzahl von Fehlern machen, ohne sie zu erkennen. Die meisten Kreditkarten-Nummern enthalten eine selbst-prüfende Zahl. Das Terminal könnte die Zahlen prüfen, bevor es ein Senden zum Zentralcomputer gestattet. Dieser Schritt würde den Zentralcomputer davor bewahren, wertvolle Verarbeitungszeit beim Prüfen der Zahl zu verschwenden.

Dies erfordert jedoch, daß das Terminal über Mittel verfügt, den Bedienenden über den Fehler zu informieren, vielleicht durch Blinken einer der Anzeigen oder durch einen anderen speziellen Anzeiger, den der Bedienende sicher bemerken kann.

Ein weiteres Problem liegt noch darin, wie der Bedienende erfährt, daß eine Eingabe verlorenging oder falsch verarbeitet wurde. Einige Terminals sperren einfach nach einer maximalen Zeitverzögerung. Der Bedienende bemerkt, daß die Besetzt-Lampe ausgegangen ist, ohne daß eine Antwort empfangen wurde. Es wird dann vom Bedienenden erwartet, daß er eine neuerliche Eingabe versucht. Nach ein oder zwei neuerlichen Eingaben sollte der Bedienende den Fehler der Service-Stelle melden.

Zahlreiche Geräte-Fehler sind ebenfalls möglich. Neben den Anzeigen, Tastatur und Prozessor existiert nun das Problem von Übermittlungs-Fehlern, sowie Fehler des Zentralcomputers.

Die Datenübertragung muß wahrscheinlich eine Fehlerprüfung und Korrektur-Verfahren einschließen. Einige Möglichkeiten sind:

KORREKTUR VON ÜBERTRAGUNGS-FEHLERN

1) Die Parität ermöglicht eine Fehler-Feststellung, jedoch keine Korrektur. Der Empfänger wird irgendeinen Weg für die Anforderung einer neuerlichen Übertragung benötigen und der Sender muß eine Kopie der Daten aufbewahren, bis ein ordnungsgemäßer Empfang bestätigt wurde. Paritätsprüfung ist jedoch leicht zu realisieren.
2) Kurze Mitteilungen können anspruchsvollere Schemas verwenden. Zum Beispiel könnte die Ja/Nein-Antwort zum Terminal so codiert werden, daß sie eine Fehler-Feststellung und Korrektur-Möglichkeit ergibt.
3) Eine Bestätigung und eine begrenzte Anzahl neuerlicher Eingaben könnte eine Anzeige auslösen, die den Bedienenden über einen Kommunikationsfehler (Unmöglichkeit zum Transfer einer Nachricht ohne Fehler) oder Computer-Fehler (keine Reaktion auf alle Nachrichten innerhalb einer bestimmten Zeitperiode) informieren würde. Ein derartiges Schema, zusammen mit dem Lampentest würde eine ziemlich einfache Fehlerdiagnose ermöglichen.

Eine Anzeige für einen Übertragungsfehler oder für einen Fehler des Zentralcomputers sollte auch das Terminal wieder freigeben, das heißt, eine neuerliche Eingabe gestatten. Dies ist erforderlich, wenn das Terminal keine Eingaben annimmt, während eine Überprüfung im Gange ist. Das Terminal sollte auch nach einer bestimmten maximalen Zeitverzögerung wieder freigegeben werden. Bestimmte Eingaben sollten für Diagnose-Zwecke reserviert werden, d.h., bestimmte Kreditkarten-Zahlen könnten zum Prüfen der internen Arbeitsweise des Terminals und zum Testen der Anzeigen verwendet werden.

ÜBERBLICK ÜBER DIE DEFINITION DER AUFGABE

Aufgaben-Definition ist ein ebenso wichtiger Teil der Software-Entwicklung, da es nur eine andere Form der Entwicklungs-Tätigkeit ist. Beachten Sie, daß dies keine Programmier-Kenntnisse oder Kenntnis des Computers erfordert. Es basiert stattdessen auf einem Verständnis des Systems und einem gesunden ingenieur-mäßigen Urteilsvermögen. Mikroprozessoren können eine Flexibilität bieten, die der Entwickler für eine Reihe von Eigenschaften einsetzen kann, die früher nicht verfügbar waren.

Die Aufgaben-Definition ist unabhängig von einem speziellen Computer, Computersprache oder Entwicklungssystem. Sie sollte jedoch die Richtlinien liefern, welche Geschwindigkeit der Computer für eine bestimmte Anwendung benötigt, und welche Art von Hardware/Software-Überlegungen der Entwickler anstellen kann. Die Phase der Aufgaben-Definition ist in der Tat unabhängig davon, ob überhaupt ein Computer verwendet wird, obwohl eine Kenntnis der Möglichkeiten des Computers dem Entwickler helfen kann, sich ein Bild über die möglichen Verfahren zu machen.

PROGRAMM-ENTWICKLUNG

Programm-Entwicklung ist die Stufe, in der die Aufgaben-Definition in Form eines Programmes formuliert wird. Wenn das Programm klein und einfach ist, kann diese Stufe vielleicht nicht mehr als die Aufstellung eines Flußdiagrammes mit nur einer Seite beinhalten. Wenn das Programm größer oder komplexer ist, sollte der Entwickler anspruchsvollere Methoden in Betracht ziehen.

Wir werden das Aufstellen von Flußdiagrammen, modulare Programmierung, strukturierte Programmierung und die sogenanne "Top-Down"-Entwicklung (Verfahren der schrittweisen Verfeinerung) besprechen. Wir werden versuchen, die Gründe für die Anwendung dieser Methoden aufzuzeigen, sowie ihre Vorteile und Nachteile. Wir werden jedoch keine Entscheidung bezüglich einer speziellen Methode fällen, da es keinen Grund gibt, daß eine Methode immer allen anderen überlegen ist. Sie sollten sich vor Augen halten, daß die Aufgabe darin besteht, ein ordnungsgemäß arbeitendes System zu schaffen und nicht einseitig einer bestimmten Methode zu folgen.

GRUNDLEGENDE PRINZIPIEN DER PROGRAMM-ENTWICKLUNG

Alle Verfahren besitzen jedoch einige offensichtliche gemeinsame Prinzipien. Viele von diesen sind die gleichen Prinzipien, die für jede Art einer Entwicklung gelten, wie etwa:

1) Gehen Sie in kleinen Schritten vor. Versuchen Sie nicht zu viel auf einmal zu erledigen.
2) Teilen Sie große Aufgaben in kleine logisch getrennte Aufgaben ein. Machen Sie die Unter-Aufgaben soweit wie möglich unabhängig von anderen, so daß sie separat getestet und daher Änderungen ohne Beeinflussung der anderen ausgeführt werden können.
3) Machen Sie den Fluß der Steuerung so einfach wie möglich, damit Fehler leichter aufzufinden sind.
4) Verwenden Sie so häufig wie möglich bildliche oder grafische Beschreibungen. Man kann sich diese leichter vorstellen als wörtliche Beschreibung. Dies ist der große Vorteil von Flußdiagrammen.
5) Achten Sie vor allem auf Klarheit und Einfachheit. Sie können die Eigenschaften verbessern (falls erforderlich), sobald das System funktioniert.
6) Gehen Sie in gründlicher und systematischer Weise vor. Verwenden Sie Prüflisten und Standard-Verfahren.
7) Fordern Sie das Schicksal nicht heraus. Verwenden Sie entweder nicht Methoden, mit denen Sie nicht völlig vertraut sind, oder verwenden Sie diese sehr sorgfältig. Achten Sie auf Situationen, die Verwirrung hervorrufen könnten und klären Sie diese so bald wie möglich.
8) Beachten Sie, daß im System nach Fehlern gesucht werden muß, dieses getestet und gewartet werden muß. Planen Sie auch für diese späteren Stufen.
9) Verwenden Sie einfache und gleichbleibende Terminologie und Methoden. Wiederholung ist kein Fehler in der Programm-Entwicklung, noch ist die Komplexität eine Tugend.
10) Formulieren Sie die Entwicklung vollständig, bevor Sie mit der Codierung beginnen. Widerstehen Sie der Versuchung mit dem Niederschreiben von Befehlen zu beginnen. Es hat nicht mehr Sinn als eine Teile-Liste aufzustellen oder einen Printplan zu zeichnen bevor Sie genau wissen, was in dem System alles enthalten ist.
11) Seien Sie besonders sorgfältig bei Faktoren, die sich ändern können. Machen Sie wahrscheinliche Änderungen so einfach wie möglich.

AUFSTELLEN VON FLUSSDIAGRAMMEN

Das Aufstellen von Flußdiagrammen ist sicherlich von allen Programmentwicklungs-Verfahren am besten bekannt. Programmierbücher beschreiben, wie Programmierer zuerst vollständige Flußdiagramme aufstellen und dann mit dem Schreiben des tatsächlichen Programmes beginnen. In der Tat arbeiten jedoch wenige Programmierer auf diese Weise und das Aufstellen der Flußdiagramme wurde häufig mehr zum Spaß gemacht. Wir wollen versuchen, sowohl die Vorteile wie die Nachteile von Flußdiagrammen zu beschreiben, und die Rolle dieser Technik in der Programm-Entwicklung zu zeigen.

VORTEILE VON FLUSSDIAGRAMMEN

Der grundlegende Vorteil des Flußdiagrammes besteht in der bildlichen Darstellung der Aufgabe. Häufig findet man, daß derartige Darstellungen mehr aussagen als geschriebene Der Entwickler kann sich das gesamte System vorstellen und die Beziehungen zwischen den einzelnen Teilen erkennen. Logische Fehler und logische Widersprüche sind oft deutlicher zu erkennen. **Im besten Falle ist das Flußdiagramm ein Bild des gesamten Systems.**

Einige speziellere Vorteile von Flußdiagrammen sind:

1) Es existieren Standard-Symbole (siehe Bild 13-7), so daß diese allen Anwendern bekannt sind.
2) Flußdiagramme können von jedem auch ohne Programmier-Kenntnisse verstanden werden.
3) Flußdiagramme können zur Aufteilung des gesamten Projekts in Unter-Aufgaben verwendet werden. Das Flußdiagramm kann dann zur Kontrolle des gesamten Fortschrittes verwendet werden.
4) Flußdiagramme zeigen die Sequenzen der Operationen und können daher eine Hilfe bei der Lokalisierung von Fehlerquellen sein.
5) Das Aufstellen von Flußdiagrammen ist ein Verfahren, das in weitem Maße auch neben der Programmierung verwendet wird.

NACHTEILE DER FLUSSDIAGRAMME

Diese Vorteile sind alle wichtig. Es ist keine Frage, daß das Aufstellen von Flußdiagrammen auch weiterhin eine sehr wesentliche Entwicklungstechnik ist. **Aber wir sollten uns auch einige der Nachteile der Flußdiagramme als Entwicklungs-Verfahren ansehen, zum Beispiel:**

1) Flußdiagramme sind schwierig zu entwickeln, zu zeichnen oder zu ändern, außer in den einfachsten Situationen.
2) Es gibt keinen einfachen Weg, um ein Flußdiagramm fehlerfrei zu machen oder zu testen.
3) Flußdiagramme können auch verwirrend sein. Entwickler finden es schwierig, ein Mittelmaß zwischen der Anzahl der benötigten Details, die ein Flußdiagramm wertvoll machen, und der Menge, die das Flußdiagramm nur wenig besser als eine Programm-Auflistung macht, zu finden.
4) Flußdiagramme zeigen nur die Programm-Organisation. Sie zeigen nicht die Organisation der Daten oder die Struktur der Eingabe/Ausgabe-Module.
5) Flußdiagramme helfen nicht bei Hardware- oder zeitlichen Problemen oder geben Hinweise, wo diese Probleme auftreten können.

Daher ist das Flußdiagramm ein wertvolles Hilfsmittel, das aber nicht weit ausgedehnt werden sollte. Flußdiagramme sind nützlich als Programm-Dokumentation, da sie Standardformen besitzen und auch für Nicht-Programmierer verständlich sind. Als Entwicklungs-Hilfsmittel jedoch können Flußdiagramme nicht mehr als eine anfängliche Richtlinie bieten. Der Programmierer kann ein detailliertes Flußdiagramm nicht von Fehlern befreien und das Flußdiagramm ist manchmal schwieriger aufzustellen als das Programm selbst.

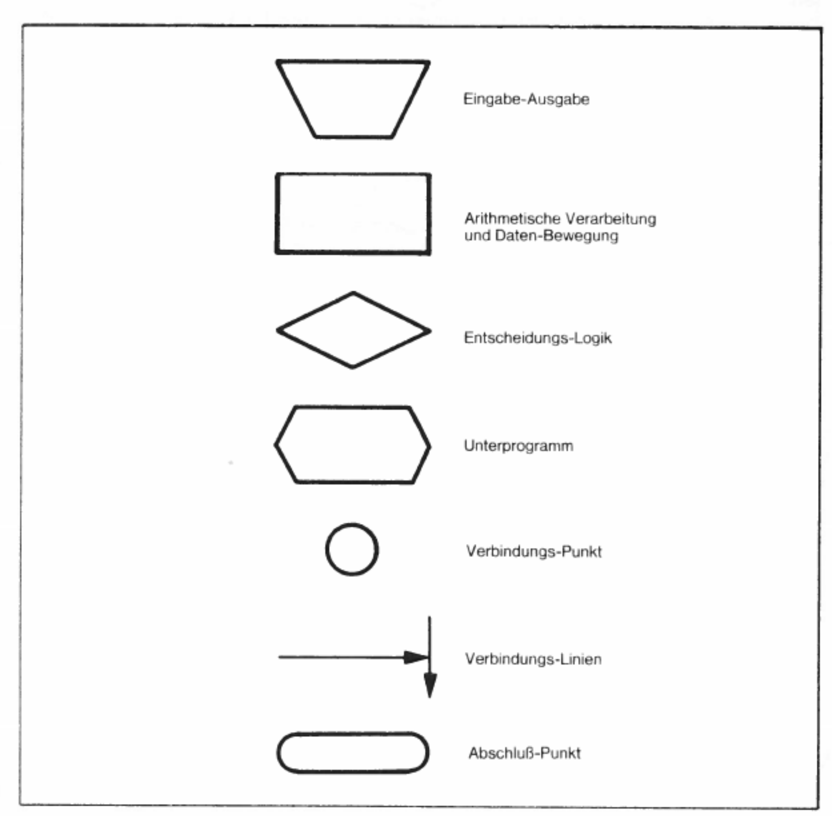


Bild 13-7. Standard-Symbole für Flußdiagramme.

BEISPIELE

Reaktion auf einen Schalter

FLUSSDIAGRAMM DES SCHALTER- UND LAMPENSYSTEMS

Für diese einfache Aufgabe, bei der ein einzelner Schalter eine Lampe eine Sekunde lang einschaltet, ist ein Flußdiagramm leicht aufzustellen.
In der Tat sind derartige Aufgaben typische Beispiele für Bücher über Flußdiagramme, obwohl sie nur einen kleinen Teil der meisten Systeme bilden. Die Datenstruktur ist hier so einfach, daß sie sicher ignoriert werden kann.

Bild 13-8 stellt das entsprechende Flußdiagramm dar. Es ist auch nicht schwierig zu entscheiden, wie weit ins Detail gegangen werden soll. Das Flußdiagramm gibt ein einfaches Bild des Verfahrens, das für jedermann verständlich ist.

Beachten Sie, daß die nützlichsten Flußdiagramme Programm-Variablen ignorieren und Fragen direkt stellen können. Natürlich sind hier häufig Kompromisse erforderlich. **Zwei Arten der Flußdiagramme sind manchmal nützlich: Eine allgemeine Version in Alltagssprache, die für Nicht-Programmierer nützlich sein wird, und eine Version für Programmierer in Ausdrücken der Programm Variablen, die für andere Programmierer von Nutzen sein werden.**

DATEN-FLUSSDIAGRAMME

Eine dritte Art von Flußdiagrammen, ein sogenanntes Daten-Flußdiagramm, kann ebenfalls sehr nützlich sein. Dieses Flußdiagramm dient als Querverweis für die anderen Flußdiagramme, da es zeigt, wie das Programm spezielle Daten-Typen handhabt. Gewöhnliche Flußdiagramme zeigen den Verlauf des Programmes, wobei unterschiedliche Arten von Daten an verschiedenen Stellen verarbeitet werden. Daten-Flußdiagramme zeigen andererseits, wie spezielle Arten von Daten das System passieren, und sich von einem Teil des Programmes zu einem anderen bewegen. Derartige Flußdiagramme sind sehr nützlich bei der Fehlersuche und Wartung, da sich Fehler häufig als falsche Behandlung von Daten erweisen.

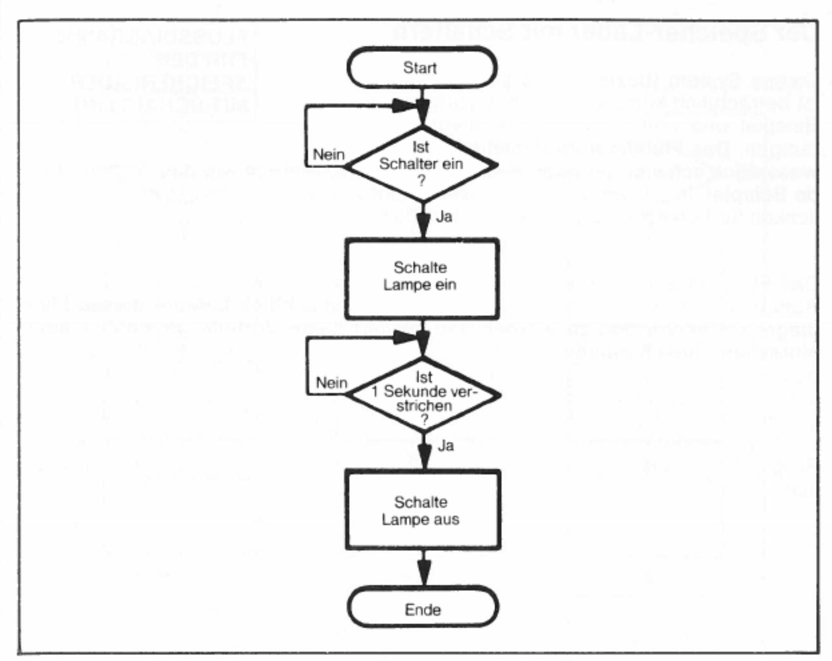


Bild 13-8. Flußdiagramm der 1-Sekunden-Reaktion auf einen Schalter.

Der Speicher-Lader mit Schaltern

FLUSSDIAGRAMM FÜR DEN SPEICHERLADER MIT SCHALTERN

Dieses System (beziehen Sie sich auf Bild 13-3) ist beträchtlich komplexer als das vorhergehende Beispiel und enthält wesentlich mehr Entscheidungen. **Das Flußdiagramm (siehe Bild 13-9) ist wesentlich schwieriger aufzustellen und nicht so einfach wie das vorhergehende Beispiel.** In diesem Beispiel stossen wir auf das Problem, daß es keine Möglichkeit für Fehlersuche und Testen des Flußdiagrammes gibt.

Das Flußdiagramm in Bild 13-9 enthält die Verbesserungen, die wir als Teil der Aufgaben-Definition vorgeschlagen haben. **Offensichtlich beginnt dieses Flußdiagramm verwirrend zu werden und verliert seine Vorteile gegenüber einer einfachen Beschreibung.** Hinzufügen anderer Möglichkeiten, die die Bedeutung der Eingabe mit Status-Lampen definieren und dem Bedienenden gestatten, die Eingabe nach deren Beendigung zu prüfen, würde das Flußdiagramm noch komplexer machen. Das Aufstellen des Flußdiagrammes aus einfachen Notizen könnte rasch eine beträchtliche Arbeit werden. Jedoch sobald das Programm geschrieben wurde, ist das Flußdiagramm als Dokumentation nützlich.

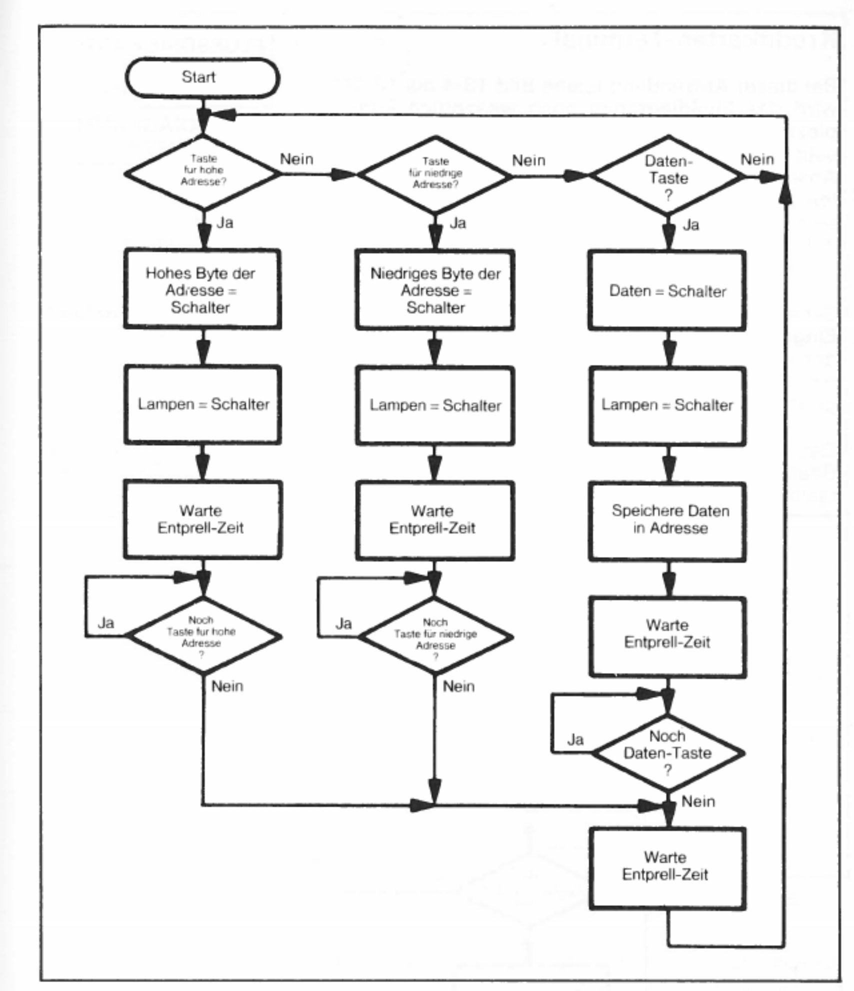


Bild 13-9. Flußdiagramm des Speicherladers mit Schaltern.

Kreditkarten-Terminal

FLUSSDIAGRAMM DER KREDIT-ÜBERPRÜFUNG

FLUSSDIAGRAMM-ABSCHNITTE

Bei dieser Anwendung (siehe Bild 13-4 bis 13-6) wird das Flußdiagramm noch wesentlich komplexer als bei dem Speicher-Lader mit Schaltern sein. Hier wäre die beste Möglichkeit, für die Abschnitte separate Flußdiagramme aufzustellen, so daß die Flußdiagramme noch handlich bleiben. Jedoch das Vorhandensein von Datenstrukturen (wie die mehrstellige Anzeige und die Nachrichten) macht die Lücke zwischen dem Flußdiagramm und dem eigentlichen Programm noch größer.

Sehen wir uns einige der Abschnitte an. **Bild 13-10 zeigt das Tastatur-Eingabe-Verfahren für die Zifferntasten.** Das Programm muß die Daten nach jeder Abtastung holen und die Ziffer in die Anzeige-Anordnung plazieren, wenn hier Platz ist. Wenn bereits zehn Ziffern in der Anzeige liegen, ignoriert das Programm einfach die Eingabe.

Das tatsächliche Programmm muß die Anzeigen zur gleichen Zeit handhaben. Beachten Sie, daß entweder die Software oder die Hardware den Tastatur-Abtastimpuls de-aktivieren muß, nachdem der Prozessor eine Ziffer gelesen hat.

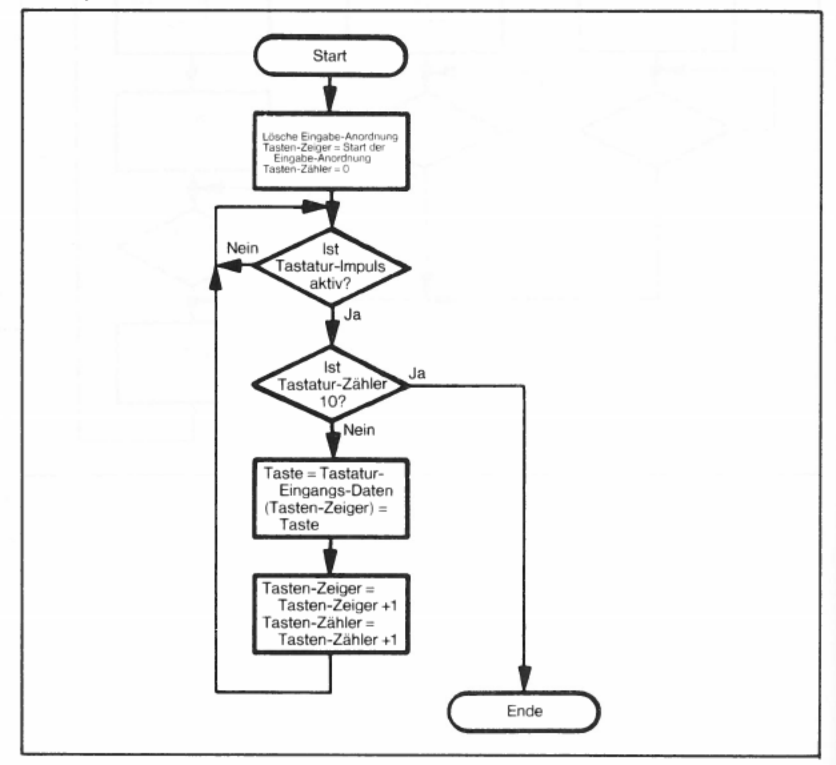


Bild 13-10. Flußdiagramm des Tastatur-Eingabe-Vorgangs.

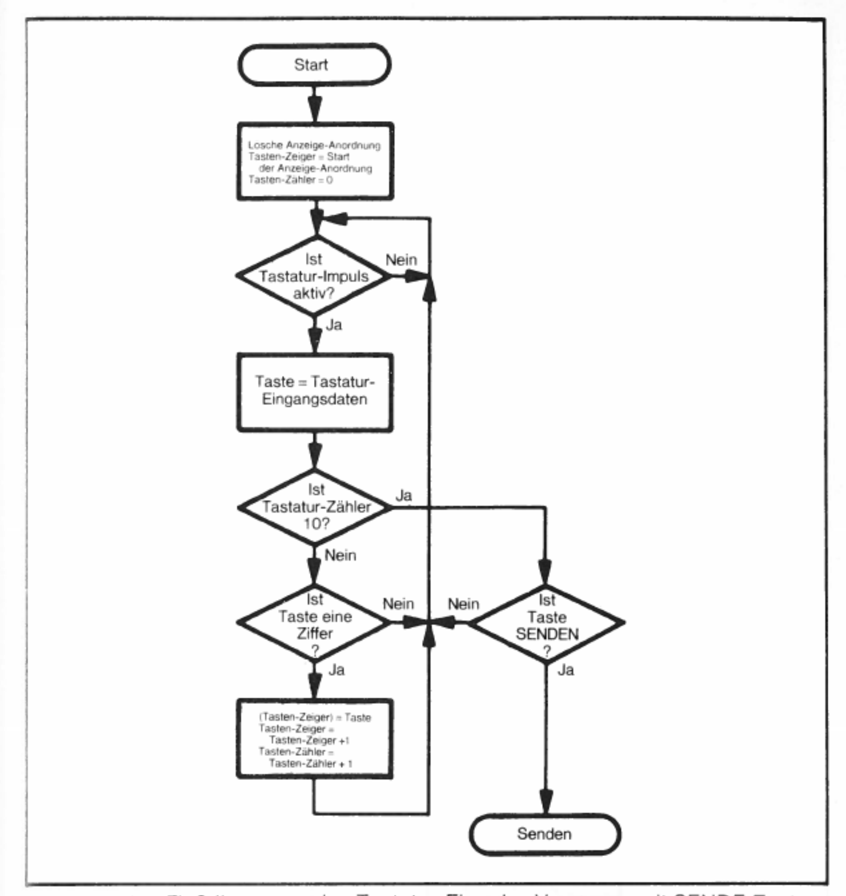


Bild 13-11. Flußdiagramm des Tastatur-Eingabe-Vorgangs mit SENDE-Taste.

In Bild 13-11 wurde die Sende-Taste hinzugefügt. Diese Taste ist natürlich wahlweise. Das Terminal könnte die Daten ebenso senden, wenn der Bedienende die vollständige Zahl eingegeben hat. Dieses Verfahren würde dem Operator jedoch keine Möglichkeit geben, die gesamte Eingabe zu prüfen. Das Flußdiagramm mit der Sendetaste ist komplexer, da es hier zwei Alternativen gibt:

1) Wenn der Bedienende nicht zehn Ziffern eingegeben hat, muß das Programm die Sendetaste ignorieren und irgendeine andere Taste in die Eingabe plazieren.
2) Wenn der Operator zehn Stellen eingegeben hat, muß das Programm auf die Sendetaste reagieren, indem es die Steuerung zur Sende-Routine überträgt und alle anderen Tasten ignoriert.

Beachten Sie, daß das Flußdiagramm wesentlich schwieriger zu organisieren und zu verfolgen ist. Es gibt auch keinen offensichtlichen Weg, um das Flußdiagramm zu überprüfen.

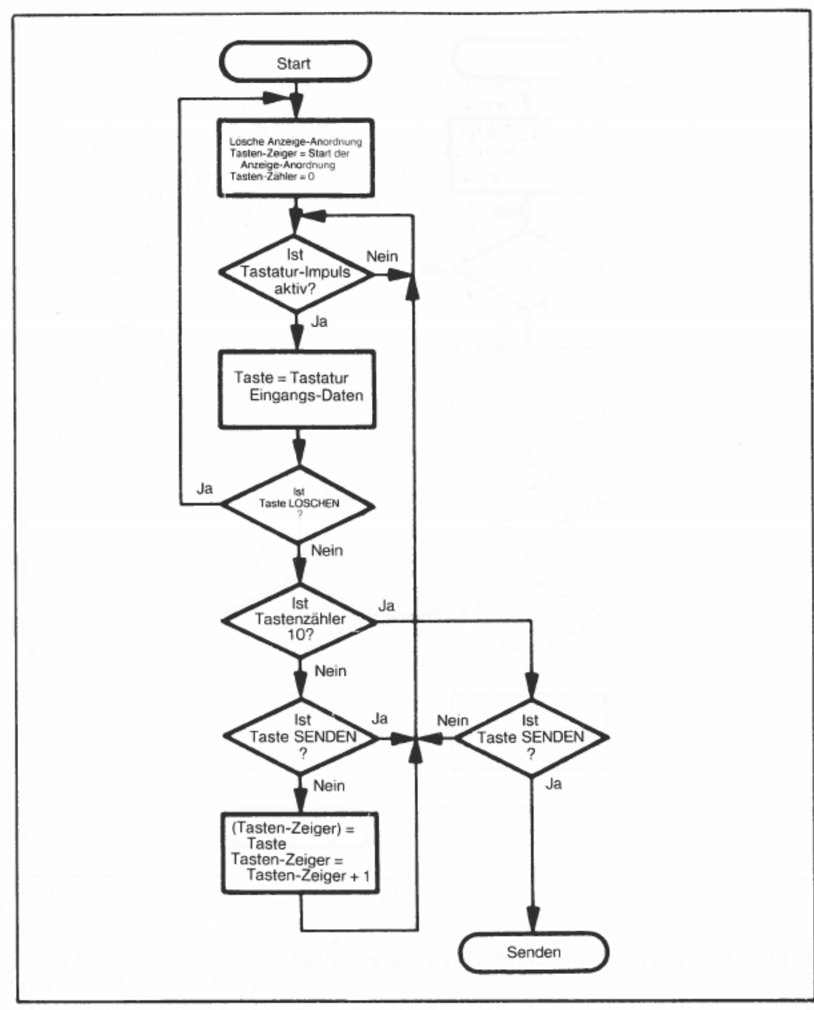


Bild 13-12. Flußdiagramm des Tastatur-Eingabe-Vorgangs mit Funktions-Tasten.

Bild 13-12 zeigt das Flußdiagramm des Tastatur-Eingabeprozesses mit allen Funktionstasten. In diesem Beispiel ist der Fluß der Steuerung keinesfalls einfach. Offensichtlich sind einige geschriebene Erklärungen erforderlich. Die Organisation und die Anordnung von komplexen Flußdiagrammen erfordert eine sorgfältige Planung. Wir sind dem Vorgang gefolgt, indem wir Eigenschaften dem Flußdiagramm einzeln beigefügt haben, aber dies resultiert noch in einer beträchtlichen Arbeit für das Neu-Zeichnen. Wieder sollten wir uns daran erinnern, daß während des Tastatur-Eingabe-Verfahrens das Programm auch die Anzeigen auffrischen muß, falls sie gemultiplext sind und nicht durch ein Schieberegister oder andere Hardware gesteuert werden.

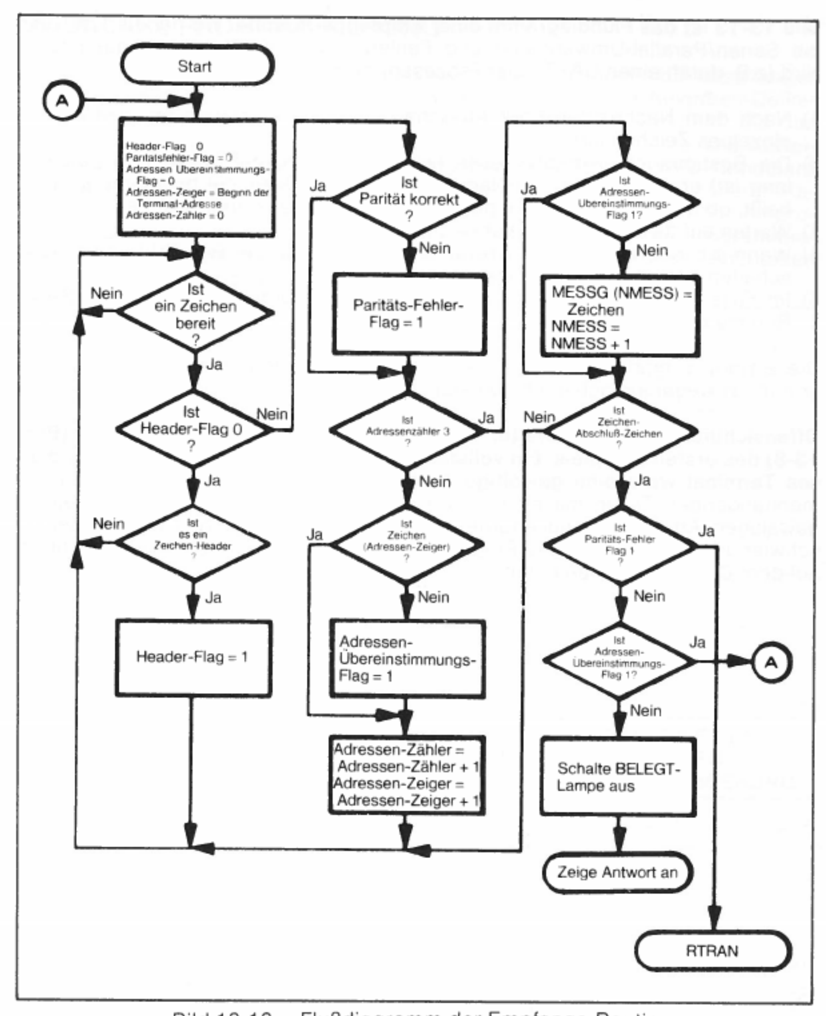


Bild 13-13. Flußdiagramm der Empfangs-Routine.

Bild 13-13 ist das Flußdiagramm einer Empfangs-Routine. Wir nehmen an, daß die Serien/Parallel-Umwandlung und Fehlerprüfung mit Hardware ausgeführt wird (z.B. durch einen UART). Der Prozessor muß:

1) Nach dem Nachrichten-Kopf Ausschau halten (wir nehmen an, daß es ein einzelnes Zeichen ist).
2) Die Bestimmungs-Adresse lesen (wir nehmen an, daß diese drei Zeichen lang ist) und sehen, ob die Nachricht für dieses Terminal bestimmt ist, daß heißt, ob die drei Zeichen mit der Terminal-Adresse übereinstimmen.
3) Warten auf das Abschluß-Zeichen.
4) Wenn die Nachricht für das Terminal bestimmt ist, die Besetzt-Lampe ausschalten und zur Antwort-Anzeige-Routine gehen.
5) Im Falle von Fehlern neuerliche Übertragung anfordern, indem zur RTRAN-Routine gegangen wird.

Diese Routine enthält eine große Anzahl von Entscheidungen und das Flußdiagramm ist weder einfach noch übersichtlich.

Offensichtlich war es ein weiter Weg von dem einfachen Flußdiagramm (Bild 13-8) des ersten Beispiels. Ein vollständiger Satz von Flußdiagrammen für dieses Terminal wäre eine gewaltige Aufgabe. Es würde aus mehreren zusammenhängenden Tafeln mit einer komplexen Logik bestehen und würde einen gewaltigen Arbeitsaufwand erfordern. Derartige Anstrengungen wären ebenso schwierig wie ein vorläufiges Programm und nicht so nützlich, da Sie es nicht auf dem Computer prüfen könnten.

MODULARE PROGRAMMIERUNG

Sobald ein Programm groß und komplex wird, ist ein Flußdiagramm nicht länger ein befriedigendes Hilfsmittel für die Entwicklung. Jedoch die Aufgaben-Definition und das Flußdiagramm kann Ihnen einige Vorstellung vermitteln, wie das Programm in vernünftige Unter-Aufgaben aufgeteilt werden könnte. **Die Aufteilung der gesamten Aufgabe in Unter-Aufgaben oder Module wird "modulare Programmierung" genannt.** Natürlich wären die meisten der in den früheren Kapiteln vorgestellten Programme typische Module in einem großen Systemprogramm. **Die Probleme, denen der Entwickler bei der modularen Programmierung gegenübersteht bestehen darin, wie das Programm in Module aufzuteilen ist und wie er die Module zusammensetzen soll.**

VORTEILE VON MODULARER PROGRAMMIERUNG

Die Vorteile der modularen Programmierung sind offensichtlich:

1) Ein einzelnes Modul ist leichter zu schreiben, fehlerfrei zu machen und zu testen als ein ganzes Programm.
2) Ein Modul ist wahrscheinlich an mehreren Stellen nützlich, sowie in anderen Programmen, besonders wenn es einigermaßen allgemein aufgebaut ist und auch allgemeine Aufgaben ausführt. Sie können sich eine ganze Bibliothek von Standard-Modulen aufbauen.
3) Modulare Programmierung gestattet dem Programmierer die Aufteilung von Aufgaben und die Verwendung früher geschriebener Programme.
4) Änderungen können an einem einzelnen Modul leichter ausgeführt werden als in dem gesamten System.
5) Fehler können häufig bei einem einzelnen Modul leichter isoliert und beseitigt werden.
6) Modulare Programmierung ergibt eine bessere Vorstellung, wieviel Fortschritt erzielt wurde und wieviel Arbeit noch zu erledigen ist.

NACHTEILE DER MODULAREN PROGRAMMIERUNG

Die Idee der modularen Programmierung ist so einleuchtend, daß die Nachteile häufig ignoriert werden. Diese sind:

1) Das Zusammensetzen der Module kann häufig ein größeres Problem sein, speziell wenn verschiedene Leute die Module geschrieben haben.
2) Module erfordern eine sehr sorgfältige Dokumentation, da sie andere Teile des Programmes beeinflussen können, wie etwa Daten-Strukturen, die von allen Modulen verwendet werden.
3) Das Überprüfen und Beseitigen von Fehlern in separaten Modulen ist schwierig, da andere Module Daten erzeugen können, die von den zu prüfenden Modulen verwendet werden, wobei wiederum andere Module die Ergebnisse verwenden können. Sie müssen möglicherweise spezielle Programme schreiben (genannt "Driver" = Treiber), nur um Datenbeispiele zu erzeugen und die Programme zu testen. Diese "Treiber" benötigen einen zusätzlichen Programmier-Aufwand, der keinen Beitrag zum Gesamtsystem leistet.

4) Programme können manchmal sehr schwierig auf vernünftige Weise zu unterteilen sein. Wenn Sie ein Programm schlecht unterteilen, so wird die Zusammensetzung sehr schwierig, da nahezu alle Fehler und die sich ergebenen Änderungen mehrere Module betreffen werden.
5) Modulare Programmierung benötigt oft zusätzliche Zeit und Speicher, da seperate Module Funktionen wiederholen können.

Deshalb besitzt die modulare Programmierung einige Nachteile, wenn sie auch eine bestimmte Verbesserung gegenüber dem einfachen Versuch zum Schreiben des gesamten Programmes auf einmal darstellt.

Wichtige Überlegungen beinhalten die Beschränkung der Menge von Informationen, die mehrere Module gleichzeitig verarbeiten, wobei Entscheidungen zu treffen sind, ob die Änderung in einem einzelnen Modul und der Zugriff eines Moduls auf ein anderes beschränkt werden soll.[5]

GRUNDLAGEN DER MODULARISIERUNG

Ein offensichtliches Problem besteht darin, daß es keine erprobten systematischen Methoden zum Modularisieren von Programmen gibt. Wir sollten daher folgende Richtlinien befolgen:[4]

1) Module, die sich auf gemeinsame Daten beziehen, sollten Teile des gleichen übergeordneten Moduls sein.
2) Zwei Module, bei denen das erste das zweite verwendet, oder von diesem abhängt, jedoch nicht umgekehrt, sollten getrennt werden.
3) Ein Modul das von mehr als einem anderen Modul verwendet wird, sollte eher Teil eines unterschiedlichen übergeordneten Moduls sein, als Teil von anderen.
4) Zwei Module, bei denen das erste von mehreren anderen Modulen und das zweite von wenigen anderen Modulen verwendet wird, sollten voneinander getrennt werden.
5) Zwei Module, deren Häufigkeit der Verwendung sich wesentlich voneinander unterscheiden, sollten Teil verschiedener Module sein.
6) Die Struktur oder Organisation von verwendeten Daten sollten in einem einzelnen Modul enthalten sein.

Wenn ein Programm schwierig zu modularisieren ist, können Sie eventuell die entsprechenden Aufgaben neu definieren. Zu viele spezielle Fälle oder zu viele Variablen, die spezielle Verarbeitung erfordern, sind typische Beispiele für eine unzureichende Aufgaben-Definition.

BEISPIELE

Reaktion auf einen Schalter

MODULARISIERUNG DES SCHALTER- UND LAMPENSYSTEMS

Dieses einfache Programm kann in zwei Module aufgeteilt werden:
Modul 1 wartet auf das Einschalten des Schalters und schaltet als Reaktion hierauf die Lampe ein.

Modul 2 liefert die Verzögerung für 1 Sekunde.

Modul 1 ist wahrscheinlich sehr speziell für dieses System, da es davon abhängt, wie der Schalter und die Lampe angeschlossen ist. Modul 2 wird allgemein verwendbar sein, da zahlreiche Aufgaben Verzögerungen benötigen. Natürlich wäre es sehr vorteilhaft, ein Standard-Verzögerungsmodul zu besitzen, das Verzögerungen unterschiedlicher Länge liefern könnte. Das Modul benötigt eine sorgfältige Dokumentation, so daß Sie wissen, wie die Länge der Verzögerung zu spezifizieren ist, wie das Modul aufzurufen ist und welche Register und Speicherplätze das Modul beeinflußt.

Eine allgemeine Version von Modul 1 wäre weit weniger nützlich, da Sie immer mit verschiedenen Arten und Verbindungen von Schaltern und Lampen zu tun haben werden.

Sie werden es wahrscheinlich einfacher finden, ein Modul für eine spezielle Konfiguration von Schaltern und Lampen zu schreiben, als den Versuch zu unternehmen, eine Standard-Routine zu verwenden. Beachten Sie den Unterschied zwischen dieser Situation und Modul 2.

Der Speicher-Lader mit Schaltern

MODULARISIERUNG DES SPEICHER-LADERS MIT SCHALTERN

Der Speicher-Lader mit Schaltern ist schwierig zu modularisieren, da alle Programm-Aufgaben von der Hardware-Konfiguration abhängen und die Aufgaben zu einfach sind, daß Module wertvoll erscheinen.
Das Flußdiagramm in Bild 13-9 könnte darauf hindeuten, daß ein Modul dasjenige wäre, das darauf wartet, daß der Bedienende eine der drei Drucktasten betätigt.

Einige andere Module könnten sein:

1) Ein Verzögerungs-Modul, das die erforderliche Verzögerung zum Entprellen der Schalter liefert.
2) Ein Schalter- und Anzeige-Modul, das die Daten von den Schaltern liest und sie zu den Anzeigen sendet.
3) Ein Lampentest-Modul.

Module, die in hohem Maße von dem System abhängen, wie die beiden letzten, sind wahrscheinlich kaum von allgemeinem Nutzen. Bei diesem Beispiel würde die modulare Programmierung keine großen Vorteile bringen.

Das Verifikations-Terminal

MODULARISIERUNG DES VERIFIKATIONS-TERMINALS

Das Verifikations-Terminal andererseits ist sehr gut für modulare Programmierung geeignet. Das gesamte System kann sehr einfach in drei Hauptmodule unterteilt werden:

1) Tastatur- und Anzeige-Modul.
2) Datenübertragungs-Modul.
3) Datenempfangs-Modul.

Ein allgemeines Tastatur- und Anzeige-Modul könnte Verwendung in zahlreichen Systemen mit Tastaturen und Anzeigen finden, die Unter-Module würden Aufgaben wie etwa folgende ausführen:

1) Erkennen einer neuen Tastatur-Eingabe und Holen der Daten.
2) Löschen der Anordnung als Reaktion auf ein Tasten-Löschen.
3) Eingabe von Ziffern in den Speicher.
4) Ausschau halten nach dem Abschluß-Zeichen oder der Sende-Taste.
5) Anzeige der Ziffern.

Obwohl die Tasien-Interpretationen und die Anzahl der Stellen variieren werden, sind die grundlegenden Vorgänge der Eingabe, Datenspeicherung und Daten-Anzeige für zahlreiche Programme gleich. Tasten-Funktionen wie "Löschen" wären ebenfalls Standard. **Natürlich muß der Entwickler überlegen, welche Module in anderen Anwendungen von Nutzen sein könnten und diesen Modulen besondere Aufmerksamkeit schenken.**

Das Datenübertragungs-Modul könnte ebenfalls in Unter-Module aufgeteilt werden wie:

1) Hinzufügen des Nachrichten-Kopfes (Header).
2) Übertragung von Zeichen, wie sie die Ausgangsleitung verarbeiten kann.
3) Erzeugen von Verzögerungen zwischen Bits oder Zeichen.
4) Hinzufügen des Abschluß-Zeichens (Trailer).
5) Prüfen auf Übertragungsfehler, d.h., keine Quittierung oder keine Möglichkeit eines Senden mit Fehler.

Das Daten-Empfangsmodul könnte folgende Untermodule enthalten:

1) Ausschau halten nach dem Nachrichtenkopf-Zeichen.
2) Überprüfen der Bestimmungsadresse der Nachricht gegenüber der Terminal-Adresse.
3) Speichern und Interpretieren der Nachricht.
4) Ausschau halten nach dem Abschluß-Zeichen.
5) Erzeugen von Bit- oder Zeichen-Verzögerungen.

PRINZIP DES DEFINITIONS-HIDING

Beachten Sie wie wichtig es hier ist, daß jede Entwicklungs-Entscheidung (wie die Bit-Rate, Nachrichten-Format, oder Fehlerprüf-Verfahren) in nur einem Modul enthalten ist. Eine Änderung in einer dieser Entscheidungen wird dann nur Änderungen in diesem einzelnen Modul erfordern. Die anderen Module sollten so geschrieben werden, daß sie völlig unabhängig von den gewählten Werten oder Methoden sind, die in dem betreffenden Modul verwendet werden. Ein wichtiges Konzept ist hier das Prinzip von Pamas "information-hiding" (= etwa: Verbergen oder Verhüllen von Informationen)[5] wobei sich die Module nur Informationen teilen, die absolut notwendig zur Durchführung der Aufgabe sind. Andere Informationen sind innerhalb eines einzelnen Moduls "eingeschlossen".

Bei der Handhabung von Fehlern sollte dieses Prinzip angewendet werden. Wenn ein Modul einen tödlichen Fehler entdeckt, sollte es nicht versuchen, diesen zu handhaben. Stattdessen sollte es das aufrufende Modul über den Fehlerstatus informieren und diesem Modul die Entscheidung überlassen, wie weiter fortzufahren ist. Der Grund hierfür ist, daß das untergeordnete Modul häufig über zu wenig Informationen verfügt, um entsprechende Verfahren einzuleiten. Nehmen Sie beispielsweise an, daß das niederwertige Modul numerische Eingaben von einem Anwender aufnimmt. Dieses Modul erwartet eine Reihe von numerischen Daten, abgeschlossen durch ein Carriage-Return. Die Eingabe eines nicht-numerischen Zeichens bewirkt, daß auf abnormale Weise abgeschlossen wird. Da das Modul den Zusammenhang nicht kennt (d.h. ist die numerische Reihe ein Operand, eine Zeilen-Nummer, die Nummer einer E/A-Einheit oder die Länge einer Datei?), kann es nicht entscheiden, wie mit diesen Fehlern zu verfahren ist. Wenn ein Modul immer selbst versuchen würde, diesen Fehler zu beseitigen, würde es seine universelle Verwendbarkeit verlieren und nur in jenen Situationen von Nutzen sein, in der gerade dieses Verfahren erforderlich ist.

ÜBERBLICK ÜBER MODULARE PROGRAMMIERUNG

Modulare Programmierung kann sehr nützlich sein, wenn Sie folgende Regeln beachten:

REGELN FÜR MODULARE PROGRAMMIERUNG

1) **Verwenden Sie Module mit 20 bis 50 Zeilen.** Kürzere Module sind häufig eine Zeitverschwendung, während längere Module selten universell und schwierig zu integrieren sind.
2) **Versuchen Sie, die Module einigermaßen universell zu machen.** Unterscheiden Sie zwischen allgemeinen Eigenschaften, wie ASCII-Code oder asynchrone Übertragungs-Formate, die für zahlreiche Anwendungen und Tasten-Identifikationen gleich sein werden, sowie der Anzahl der Anzeigen oder Anzahl der Zeichen in einer Nachricht, die wahrscheinlich einmalig in einer bestimmten Anwendung vorkommen. Machen Sie das Ändern letzterer Parameter einfach. Größere Änderungen, wie unterschiedliche Zeichencodes, sollten von separaten Modulen gehandhabt werden.
3) **Nehmen Sie sich besonders Zeit für Module wie Anzeigen, Anzeigen-Verarbeitung, Tastatur-Bedienung etc., die in anderen Projekten oder in zahlreichen Stellen des momentanen Programmes nützlich sein werden.**
4) **Versuchen Sie die Module so ausgeprägt und logisch getrennt wie möglich zu halten.**
5) **Versuchen Sie nicht, einfache Aufgaben zu modularisieren,** bei denen ein Neuschreiben der geamten Aufgabe einfacher sein kann, als ein Assemblieren oder Modifizieren des Moduls.

STRUKTURIERTE PROGRAMMIERUNG

Wie gehen Sie vor, um Module scharf abgegrenzt zu halten und eine Wechselwirkung mit anderen zu vermeiden? Wie schreiben Sie ein Programm, das eine klare Sequenz von Operationen besitzt, so daß Sie Fehler aufspüren und korrigieren können? Eine Antwort ist die Verwendung eines Verfahrens, bekannt als "strukturierte Programmierung", wobei jeder Teil des Programmes aus Elementen eines begrenzten Satzes von Strukturen besteht und jede Struktur einen einzigen Eingang und einen einzigen Ausgang besitzt.

Bild 13-14 zeigt ein Flußdiagramm eines nicht strukturierten Programmes. Wenn ein Fehler im Modul B auftritt, haben wir fünf mögliche Quellen für diesen Fehler. Wir müssen nicht nur jede mögliche Sequenz überprüfen, sondern wir müssen uns auch vergewissern, daß jede Änderung, die zur Korrektur eines Fehlers gemacht wird, nicht eine der anderen Sequenzen beeinflußt. Das übliche Ergebnis ist, daß eine derartige Beseitigung von Fehlern einem Kampf mit einem Karaken ähnelt. Jedesmal, wenn Sie glauben, die Situation unter Kontrolle zu haben, begegnen Sie einem weiteren Arm.

GRUNDLEGENDE STRUKTUREN DER STRUKTURIERTEN PROGRAMMIERUNG

Die Antwort auf dieses Problem ist die Einrichtung einer klaren Sequenz von Operationen, so daß Sie Fehler isolieren können. Eine klare Sequenz verwendet Module mit einem einzigen Eingang und einen einzigen Ausgang. **Als grundlegende Module sind erforderlich:**

1) **Eine normale Sequenz,** das heißt, eine lineare Struktur, in der die Anweisungen oder Strukturen nacheinander ausgeführt werden. In der Sequenz:

   S1
   S2
   S3

   führt der Computer zuerst S1 dann S2 und als drittes S3 aus. S1, S2 und S3 können einzelne Befehle oder ein ganzes Programm sein.

2) **Eine bedingte Struktur.**
   Eine gebräuchliche Struktur wäre "wenn C dann S1, sonst S2", wobei C eine Bedingung ist und S1 und S2 Anweisungen oder Sequenzen. Der Cmputer führt S1 aus, wenn C wahr ist und S2, wenn C falsch ist. Bild 13-15 zeigt die Logik dieser Struktur. Beachten Sie, daß die Struktur einen einzigen Eingang und einen einzigen Ausgang besitzt. Es gibt keinen Weg in S1 oder S2 einzutreten oder es zu verlassen, außer durch die Struktur.

3) **Eine Schleifenstruktur.**
   Die gebräuchliche Schleifenstruktur ist "erledige S solange C", wobei C eine Bedingung ist und S eine Anweisung oder eine Sequenz von Anweisungen. Der Computer prüft C und führt S aus, wenn C wahr ist. Diese Struktur (siehe Bild 13-16) besitzt ebenfalls einen einzigen Eingang und einen einzigen Ausgang. Beachten Sie, daß der Computer S überhaupt nicht ausführen wird, wenn C ursprünglich falsch war, da der Wert von C vor der Ausführung von S geprüft wird.

Bei den meisten strukturierten Programmiersprachen ist eine alternative Schleifenstruktur vorgesehen. Dieser Aufbau ist bekannt als das sogenannte "Do-until". Sein grundlegender Aufbau ist "do S until C" (führe S aus bis C), wobei C eine Bedingung und S eine Anweisung oder eine Anweisungs-Sequenz ist. Sie ähnelt dem "Do-while"-Aufbau mit der Ausnahme, daß das Prüfen der Schleifen-Bedingung C am Ende der Schleife ausgeführt wird. Dadurch wird garantiert, daß die Schleife immer wenigstens einmal ausgeführt wird. Dadurch wird garantiert, daß die Schleife immer wenigstens einmal ausgeführt wird. Dies ist im Flußdiagramm in Bild 13-17 dargestellt. Die gemeinsame index-gesteuerte oder DO-Schleife kann als ein spezieller Fall einer dieser beiden grundlegenden Schleifen-Strukturen ausgeführt werden.

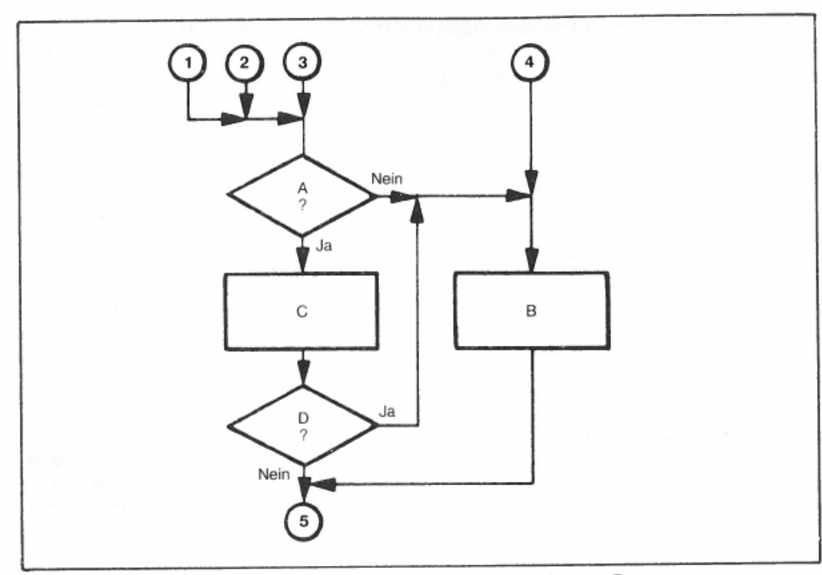


Bild 13-14. Flußdiagramm eines unstrukturierten Programms.

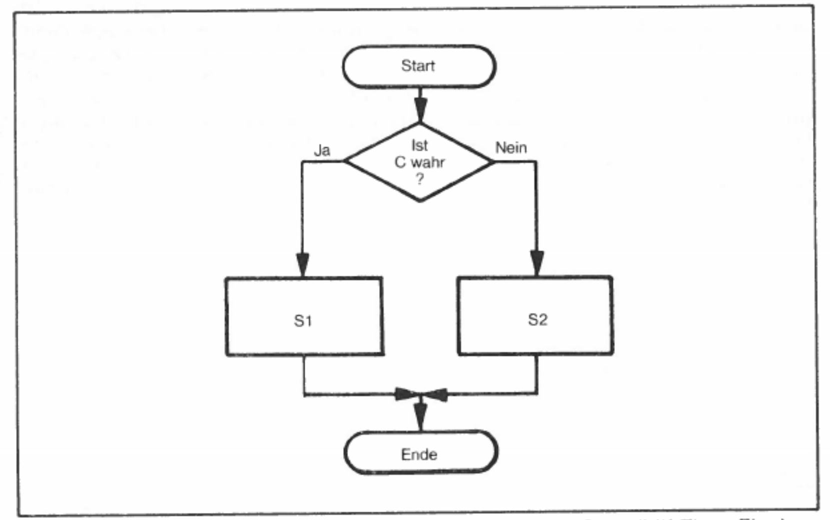


Bild 13-15. Flußdiagramm der Struktur "Wenn-Dann-Sonst" (If-Then-Else). (Verzweigung)

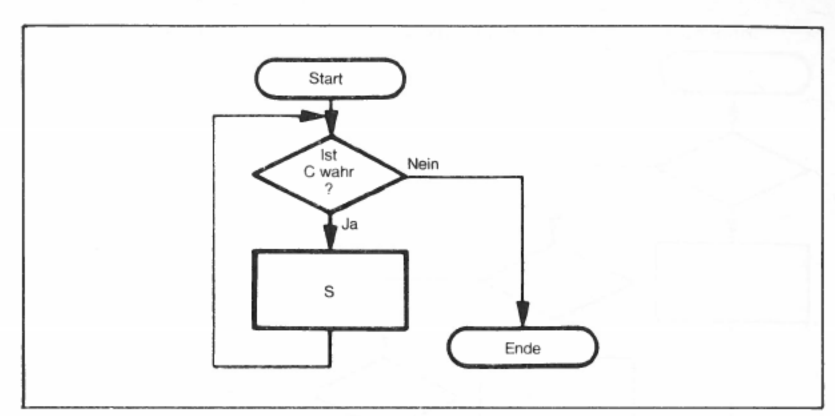


Bild 13-16. Flußdiagramm der Struktur "Führe aus-Solange" (Do-While) (Schleife)

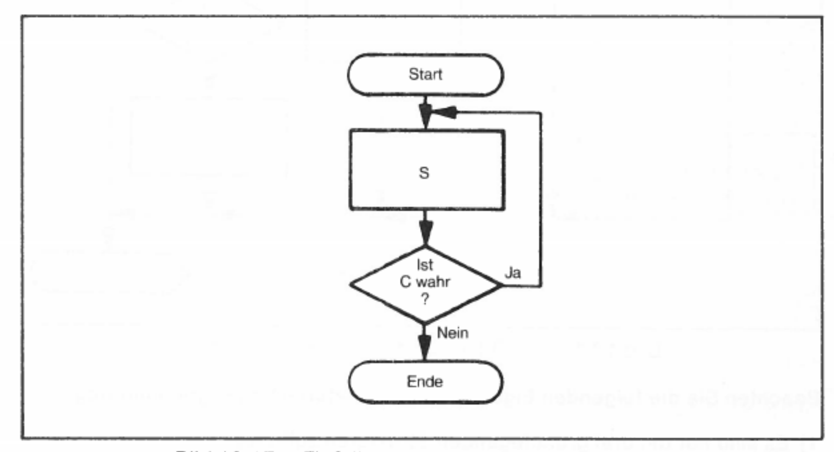


Bild 13-17. Flußdiagramm der "Do-Until"-Struktur.

4) **Eine Fall-Struktur**
Obwohl die "Fall"-Struktur nicht so einfach aufgebaut wie das sequentielle "if-then-else" und "do-while" ist, wird sie so häufig gebraucht, daß wir sie hier als Ergänzung zu den Beschreibungen der grundlegenden Strukturen hinzufügen wollen. Die Fall Struktur ist "case I of S0, S1,... Sn", wobei I ein Index und S0, S1,... Sn Anweisungen oder Anweisungs-Sequenzen sind. Wenn I gleich 0 ist, dann wird die Anweisung S0 ausgeführt, wenn I gleich 1 ist, dann wird die Anweisung S1 durchgeführt, etc. Es wird nur eine von n-Anweisungen ausgeführt. Nach seiner Ausführung geht die Steuerung zur nächsten sequentiellen Anweisung über, die der Gruppe mit den "Fall-Anweisungen" folgt. Wenn I größer als n ist (d.h. die Anzahl der Anweisungen in der "Fall"-Anweisung), dann wird keine der Anweisungen in dieser Gruppe ausgeführt und die Steuerung geht direkt zur nächsten sequentiellen Anweisung nach dieser Gruppe über. Dies wird im Flußdiagramm in Bild 13-18 dargestellt.

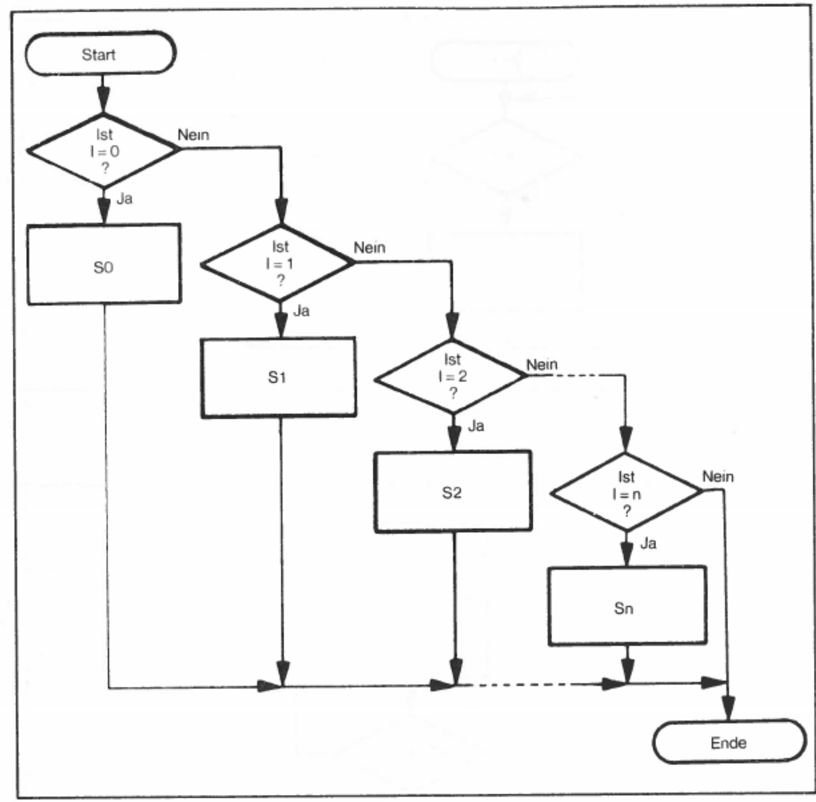


Bild 13-18. Flußdiagramm der "Case"-Struktur.

Beachten Sie die folgenden Eigenschaften strukturierter Programmierung:

1) Es sind nur die drei grundlegenden Strukturen zulässig.
2) Strukturen können auf jede Ebene der Komplexität verschachtelt werden, so daß jedes Programm wiederum jede der Strukturen enthalten kann.
3) Jede Struktur besitzt einen einzigen Eingang und einen einzigen Ausgang.

Einige Beispiele der in Bild 13-15 gezeigten bedingten Struktur sind:

BEISPIELE VON STRUKTUREN

1) S2 inbegriffen:
```
if X ≥ 0 then NPOS = NPOS + 1
   else NNEG = NNEG + 1
```

Sowohl S1 wie S2 sind einzelne Anweisungen.

2) S2 weggelassen:
```
if X ≠ 0 then Y = 1/X
```

Hier wird keine Tätigkeit ausgeführt, wenn C (X ≠ 0) falsch ist. S2 und "sonst" kann in diesem Fall weggelassen werden.

Einige Beispiele der in Bild 13-16 dargestellten Schleifenstrukturen sind:

1) Bilde die Summe der ganzen Zahlen von 1 bis N.
```
I = 0
SUM = 0
do while I < N
  I = I+1
  SUM = SUM + I
end
```

Der Computer führt die Schleife so lange aus, wie I < N. Wenn N = 0, wird das Programm innerhalb des "führe aus solange" überhaupt nicht ausgeführt.

2) Zähle Zeichen in einer Anordnung SENTENCE bis Sie einen ASCII-Punkt finden.

```
NCHAR = 0
do while SENTENCE (NCHAR) ≠ PERIOD
  NCHAR = NCHAR+1
end
```

Der Computer führt die Schleife so lange aus, wie das Zeichen in SENTENCE kein ASCII-Punkt ist. Die Zählung ist 0, wenn das erste Zeichen ein Punkt ist.

Die Vorteile der strukturierten Programmierung sind:

VORTEILE DER STRUKTURIERTEN PROGRAMMIERUNG

1) Die Sequenz der Operationen ist leicht zu verfolgen. Dadurch wird das Testen und Fehlersuchen leicht gemacht.
2) Die Anzahl der Strukturen ist begrenzt und die Terminologie ist standardisiert.
3) Die Strukturen sind leicht in Module umzuwandeln.
4) Theoretische Überlegungen haben ergeben, daß der gegebene Satz von Strukturen vollständig ist, das heißt, alle Programme können mit diesen drei Strukturen geschrieben werden.
5) Die strukturierte Version eines Programmes dokumentiert sich zum Teil selbst und ist verhältnismäßig leicht zu lesen.
6) Strukturierte Programme sind leicht mit Flußdiagrammen oder anderen grafischen Methoden zu beschreiben.
7) Strukturierte Programmierung hat sich in der Praxis zur Steigerung der Produktivität des Programmierers bewährt.

Strukturierte Programmierung zwingt den Programmierer grundsätzlich zu mehr Disziplin als modulare Programmierung. Das Ergebnis ist mehr Systematik und besser organisierte Programme.

Nachteile der strukturierten Programmierung sind:

NACHTEILE DER STRUKTURIERTEN PROGRAMMIERUNG

1) Nur einige wenige höhere Programmiersprachen (z.B. PL/M, PASCAL) werden die Strukturen direkt annehmen. Der Programmierer muß daher eine zusätzliche Übersetzungsstufe verwenden, um die Strukturen in den Assemblersprachen-Code umzuwandeln. Die strukturierte Version des Programmes ist jedoch häufig als Dokumentation sehr nützlich.
2) Strukturierte Programme werden häufig langsamer ausgeführt und benötigen mehr Speicher als unstrukturierte Programme.
3) Die Begrenzung der Strukturen auf drei grundlegende Formen macht manche Aufgaben in der Ausführung sehr mühsam. Die Vollständigkeit der Strukturen bedeutet nur, daß alle Programme mit ihnen ausgeführt werden können. Es bedeutet nicht, daß ein gegebenes Programm effizient und bequem geschaffen werden kann.
4) Die Standard-Strukturen sind häufig etwas verwirrend, d.h. verschachtelte Strukturen "wenn-dann-sonst" (if-then-else) können sehr schwierig zu lesen sein, da es keine deutliche Anzeige geben kann, wo eine hiervon endet. Eine Serie von "Führe-aus-solange"-Schleifen kann ebenfalls schwierig zu lesen sein.
5) Strukturierte Programme berücksichtigen nur die Sequenz von Programm-Operationen, nicht den Datenfluß. Daher können die Strukturen Daten manchmal sehr mühsam behandeln.
6) Wenige Programmierer sind mit der strukturierten Programmierung vertraut. Viele finden die Standard-Struktur mühsam und hemmend.

Wir wollen uns nicht als Richter über die Verwendung der strukturierten Programmierung aufschwingen. Es ist eben ein bestimmter Weg, Programm-Entwicklung systematisch durchzuführen. Im allgemeinen ist sie bei folgenden Situationen am nützlichsten:

WANN SOLLTE STRUKTURIERTE PROGRAMMIERUNG VERWENDET WERDEN

1) Größere Programme, wahrscheinlich tausend oder mehr Befehle.
2) Anwendungen, in denen die Verwendung des Speichers nicht kritisch ist.
3) Anwendungen mit kleinen Stückzahlen, wo die Kosten für die Software-Entwicklung und die Fehlersuche wichtige Faktoren sind.
4) Anwendungen, die keine komplexen Daten-Strukturen enthalten.
5) Anwendungen mit Reihen-Manipulationen, Prozess-Steuerung oder andere Algorithmen, anstatt einfacher Bit-Manipulationen.
6) Anwendungen, in denen höhere Programmiersprachen verwendet werden.

Für die Zukunft ist anzunehmen, daß die Speicherkosten weiter sinken, die durchschnittliche Länge von Mikroprozessor-Programmen steigt und die Kosten der Software-Entwicklung abnehmen. Daher werden Methoden wie die strukturierte Programmierung, die die Software-Entwicklungskosten für grössere Programme verringert, jedoch mehr Speicher benötigt, auch wertvoller werden.
Wenn die Grundlagen der strukturierten Programmierung gewöhnlich in höheren Programmiersprachen vorkommen, so bedeutet dies nicht, daß strukturierte Programmierung nicht in der Assembler-Sprachen-Programmierung anwendbar ist. **Im Gegenteil, der Assembler-Sprachen-Programmierer, der in dieser Programmiersprache weitgehende Freiheit besitzt, benötigt strukturierte Programmierung in besonders hohem Maße. Die Schaffung von Modulen mit einzelnen Eintritts- und Austritts-Punkten, die Verwendung von Steuer-Strukturen und die Möglichkeit, jedes Modul einfachst aufzubauen, macht die Codierung in Assemblersprache besonders effizient.**

BEISPIELE

Reaktion auf einen Schalter

Die strukturierte Version dieses Beispieles ist:

STRUKTURIERTE PROGRAMMIERUNG BEI DEM SCHALTER- UND LAMPENSYSTEM

```
SWITCH = OFF
do while SWITCH = OFF
  READ SWITCH
  end
LIGHT = ON
DELAY 1
LIGHT = OFF
```

EIN uns AUS müssen die entsprechnenden Definitionen für den Schalter und die Lampe besitzen. Wir nehmen an, daß VERZÖGERUNG ein Modul ist, das eine Verzögerung ergibt, die durch seine Parameter in Sekunden gegeben ist.

Eine Anweisung in einem strukturierten Programm kann in Wirklichkeit ein Unterprogramm sein. Um jedoch den Regeln der strukturierten Programmierung gerecht zu werden, darf das Unterprogramm keine anderen Ausgänge haben, als den einen, der die Steuerung zum Hauptprogramm zurückgibt.

Da "Führe aus solange" die Bedingung vor Ausführung der Schleife prüft, setzen wir vor dem Start die Variable SCHALTER auf AUS. Die strukturierte Programmierung ist übersichtlich, lesbar und leicht von Hand zu überprüfen. Wir werden jedoch wahrscheinlich etwas mehr Speicher als bei einem unstrukturierten Programm benötigen, das nicht SCHALTER initialisieren müsste und die Lese- und Prüf-Verfahren miteinander kombinieren könnte.

Der Speicher-Lader mit Schaltern

STRUKTURIERTE PROGRAMMIERUNG DES SPEICHERLADERS MIT SCHALTERN

Der Speicher-Lader mit Schaltern ist eine etwas komplexere Aufgabe für die strukturierte Programmierung. Wir können das Flußdiagramm von Bild 13-9 wie folgt darstellen (ein . zeigt einen Kommentar an):

```
.
.INITIALISIERE VARIABLEN
.
HIADDRESS = 0
LOADDRESS = 0
.
.DIESES PROGRAMM VERWENDET EINEN "FÜHRE AUS SOLANGE"-AUFBAU
.OHNE BEDINGUNG
.(GENANNT EINFACH "FÜHRE IMMER AUS") DESHALB FÜHRT DAS SYSTEM
.KONTINUIERLICH DAS PROGRAMM AUS, DAS IN SEINER "FÜHRE AUS
.SOLANGE"-SCHLEIFE ENTHALTEN IST.
.
führe immer aus
.
.TESTE AUF HIADDRESS-TASTE, FÜHRE DIE ERFORDERLICHE VERARBEI-
.TUNG AUS, WENN SIE EIN IST.
.
    if HIADDRESSBUTTON = 1 then
      begin
        HIADDRESS = SWITCHES
        LIGHTS = SWITCHES
        do
          DELAY (DEBOUNCE TIME)
        until HIADDRBUTTON ≠ 1
      end
.
.TESTE AUF LOADDRESS-TASTE, FÜHRE VERARBEITUNG DER NIEDRI-
GEN .ADRESSE AUS, WENN SIE AUF EIN IST.
.
    if LOADDRBUTTON = 1 then
      begin
        LOADDRESS = SWITCHES
        LIGHTS = SWITCHES
        do
          DELAY (DEBOUNCE TIME)
        until LOADDRBUTTON ≠ 1
      end
.
.TESTE AUF DATEN-TASTE, UND SPEICHERE DATEN IN DEN SPEICHER,
.WENN SIE AUF EIN IST.
.
    if DATABUTTON = 1 then
      begin
        DATA = SWITCHES
        LIGHTS = SWITCHES
        (HIADDRESS, LOADDRESS) = DATA
        do
          DELAY (DEBOUNCE TIME)
        until DATABUTTON ≠ 1
      end
end
```



```
.
.DAS OBIGE "END" BEENDET DIE ENDLOSE SCHLEIFE.
.
```

Strukturierte Programme sind nicht leicht zu schreiben, können jedoch einen guten Einblick in die gesamte Programmlogik geben. Sie können die Logik der strukturierten Programme von Hand prüfen, bevor sie einen tatsächlichen Code schreiben.

Das Kredit-Verifikations-Terminal

STRUKTURIERTES PROGRAMM FÜR DAS KREDIT-VERIFIKATIONS-TERMINAL

STRUKTURIERTE TASTATUR-ROUTINE

Sehen wir uns die Tastatur-Eingabe für das Terminal an. Wir wollen annehmen, daß die Anzeige-Anordnung ENTRY ist, der Tastatur-Abtast-Impuls KEYSTROBE ist und die Tastatur-Daten KEYIN. **Das strukturierte Programm ohne die Funktions-Tasten ist:**

```
NKEYS = 10
.
.LÖSCHE ENTRY FÜR DEN START
.
    do while NKEYS > 0
      NKEYS = NKEYS -1
      ENTRY (NKEYS) = 0
    end
.
.HOLE EIN VOLLSTÄNDIGES EINGABE VON DER TASTATUR
.
    do while NKEYS < 10
      if KEYSTROBE = ACTIV then
        begin
          KEYSTROBE = INACTIV
          ENTRY (NKEYS) = KEYIN
          NKEYS = NKEYS +1
      end
    end
```

Das Hinzufügen der SENDE-Taste bedeutet, daß das Programm zusätzliche Ziffern ignorieren muß, nachdem es eine vollständige Eingabe hat und die SENDE-Taste ignorieren muß, bis es eine vollständige Eingabe besitzt. Das strukturierte Programm lautet:

```
NKEYS = 10
.
.LÖSCHE EINGABE ZUM START
.
    do while NKEYS > 0
      NKEYS = NKEYS -1
      ENTRY (NKEYS) = 0
    end
```

```
.
.WARTE AUF VOLLSTÄNDIGE EINGABE UND SENDE-TASTE
.
   do while KEY ≠ SEND OR NKEYS ≠ 10
     if KEYSTROBE = AKTIV then
       begin
         KEYSTROBE = INACTIV
         KEY = KEYIN
         if NKEYS ≠ 10 AND KEY ≠ SEND then
           begin
             ENTRY (NKEYS) = KEY
             NKEYS = NKEYS+1
           end
       end
     end
```

Beachten Sie folgende Eigenschaften dieses strukturierten Programmes:

1) Das zweite "wenn-dann" ist im ersten verschachtelt, da die Tasten nur eingegeben werden, nachdem der Abtast-Impuls erkannt wurde. Wenn sich das zweite "wenn-dann" auf der gleichen Ebene wie das erste befinden würde, könnte eine einzelne Taste die Eingabe füllen, da sein Wert in der Anordnung während jeder Wiederholung der "führe -aus-solange"-Schleife eingegeben werden könnte.
2) KEY muß anfangs nicht definiert werden, da KEY auf null gesetzt wird, als Teil des Löschens der Eingabe.

Das Hinzufügen der CLEAR-Taste gestattet dem Programm das Löschen der Eingabe durch Simulieren eines Drückens von CLEAR, d.h. durch Setzen von NKEYS auf 10 und KEY auf CLEAR vor dem Start. Das strukturierte Programm muß auch nur Stellen löschen, die vorher gefüllt wurden. **Das neue strukturierte Programm lautet:**

```
.
.SIMULIERE VOLLSTÄNDIGES LÖSCHEN
.
.
NKEYS = 10
KEY = LÖSCHEN
.
. WARTE AUF VOLLSTÄNDIGE EINGABE UND SENDE-TASTE
do while KEY ≠ SEND OR NKEYS ≠ 10
.
.LÖSCHE GESAMTE EINGABE, WENN CLEAR-TASTE BETÄTIGT WIRD
   if KEY = CLEAR then
     begin
       KEY = O
       do while NKEYS > 0
         NKEYS = NKEYS – 1
         ENTRY (NKEYS) = 0
       end
     end
```



```
.
.HOLE ZIFFER WENN EINGABE UNVOLLSTÄNDIG
.
   if KEYSTROBE = INAKTIV then
     begin
       KEYSTROBE = INACTIV
       KEYSTROBE = KEYIN
       if KEY < 10 AND NKEYS ≠ 10 then

         begin
           ENTRY (NKEYS) = KEY
           NKEYS = NKEYS+1
         end
     end
end
```

Beachten Sie, daß das Programm KEY auf null nach dem Löschen der Anordnung stellt, so daß die Operation nicht wiederholt wird.

STRUKTURIERTE EMPFANGS-ROUTINE

Wir können auf ähnliche Weise ein strukturiertes Programm für die Empfangs-Routine aufstellen. Ein anfängliches Programm könnte nur nach dem Nachrichten-Kopf und den Abschlußzeichen Ausschau halten.
Wir wollen annehmen, daß das RSTB ein Indikator ist, der sagt, daß ein Zeichen bereit ist. **Das strukturierte Programm lautet:**

```
.
.LÖSCHE NACHRICHTENKOPF-FLAG ZUM START
.
HFLAG = 0
.
.WARTE AUF NACHRICHTENKOPF UND ABSCHLUSSZEICHEN
.
do while HFLAG = 0 OR CHAR ≠ TRAILER
.
.HOLE ZEICHEN WENN BEREIT, HALTE AUSSCHAU NACH NACHRICHTEN-
.KOPF
.
   if RSTB = ACTIV then
     begin
       RSTB = INACTIV
       CHAR = INPUT
       if CHAR = HEADER then HFLAG = 1
     end
```

Nun können wir den Abschnitt hinzufügen, der die Adresse der Nachrichten gegenüber den drei Ziffern im TERMINAL ADDRESS (TERMADDR) prüft. Wenn eine der entsprechenden Ziffern nicht gleich ist, dann wird das ADDRESS MATCH-Flag (ADDRMATCH) auf 1 gesetzt.

```
.
.LÖSCHE HEADER-FLAG, ADDRESS-MATCH-FLAG, ADRESSENZÄHLER FÜR
.DEN START
.
HFLAG = 0
ADDRMATCH = 0
ADDRCTR = 0
.
.WARTE AUF NACHRICHTENKOPF, BESTIMMUNGSADRESSE UND
.ABSCHLUSSZEICHEN
do while HFLAG = 0 OR CHAR ≠ TRAILER OR ADDRCTR ≠ 3
.
.HOLE ZEICHEN WENN BEREIT
.
    if RSTB = ACTIV then
      begin
        RSTB = INACTIV
        CHAR = INPUT
      end
.
.PRÜFE AUF TERMINAL-ADRESSE UND HEADER
.
    if HFLAG = 1 AND ADDRCTR ≠ 3 then
      begin
        ADDRMATCH = 1
        ADDRCTR = ADDRCTR+1
      if CHAR = HEADER then HFLAG = 1
end
```

Das Programm muß nun auf einen Nachrichtenkopf warten, einen dreistelligen Identifikationscode und auf ein Abscchlußzeichen. Sie müssen sehr sorgfältig beachten, was während der Wiederholung geschieht, wenn das Programm den Nachrichtenkopf findet und was geschieht, wenn ein fehlerhaftes Identifikations-Codezeichen gleich dem Abschlußzeichen ist.

Zusätzlich kann die Nachricht in MESSG gespeichert werden. NMESS ist die Zahl der Zeichen in der Nachricht. Wenn sie am Ende nicht null ist, weiß das Programm, daß das Terminal eine gültige Nachricht empfangen hat. Wir haben nicht versucht, die Logik-Ausdrücke in diesem Programm auf ein Minimum zu reduzieren.

```
.
.LÖSCHE FLAGS, ZÄHLER ZUM START
.
HFLAG = 0
ADDRMATCH = 0
ADDRCTR = 0
NMESS = 0
.
.WARTE AUF HEADER, BESTIMMUNGSADRESSE UND ABSCHLUSSZEICHEN
do while HFLAG = 0 OR CHAR ≠ TRAILER or ADDRCTR ≠ 3
.HOLE ZEICHEN WENN BEREIT
.
    if RSTB = ACTIV then
      begin
        RSTB = INAKTIV
        CHAR = INPUT
      end
```



```
.
.LIES NACHRICHT WENN BESTIMMUNGSADRESSE = TERMINAL-ADRESSE
.
    if HFLAG = 1 AND ADDRCTR = 3 then
      if ADDRMATCH = 0 and CHAR ≠ TRAILER then
        begin
          MESSG (NMESS) = CHAR
          NMESS = NMESS+1
        end
.
.PRÜFE AUF TERMINAL-ADRESSE
.
    if HFLAG = 1 AND ADDRCTR = 3 then
    if CHAR ≠ TERMADDR(ADDRCTR) then
      begin
        ADDRMATCH = 1
        ADDRCTR = ADDCRT+1
      end
.
.HALTE NACH NACHRICHTENKOPF AUSSCHAU
.
    if CHAR = HEADER then HFLAG = 1
end
```

Das Programm prüft nur auf den Identifikations-Code, wenn es einen Nachrichtenkopf während einer vorhergehenden Wiederholung gefunden hat. Es nimmt die Nachrichten nur an, wenn es vorher einen Nachrichtenkopf und eine vollständige, übereinstimmende Bestimmungs-Adresse gefunden hat. Das Programm muß während der Wiederholungen ordnungsgemäß arbeiten, wenn es den Nachrichtenkopf, das Abschlußzeichen und die letzte Stelle der Bestimmungs-Adresse findet. Es darf nicht versuchen, den Nachrichtenkopf mit der Bestimmungs-Adresse in Übereinstimmung zu bringen oder das Abschlußzeichen oder die letzte Stelle der Bestimmungs-Adresse in die Nachricht zu plazieren. **Sie könnten versuchen, den Rest der Logik vom Flußdiagramm (Bild 13-13) dem str vom Flußdiagramm (Bild 13-13) dem strukturierten Programm hinzuzufügen. Beachten Sie, daß die Reihenfolge der Operationen häufig kritisch ist. Sie müssen sichergehen, daß das Programm nicht eine Phase abschließt und die nächste während derselben Wiederholung beginnt.**

ÜBERBLICK ÜBER DIE STRUKTURIERTE PROGRAMMIERUNG

Strukturierte Programmierung bringt zwangsläufig eine entsprechende Ordnung in die Programm-Entwicklung. Sie sind gezwungen, die Arten der von Ihnen verwendeten Strukturen zu begrenzen, sowie die Sequenz der Operationen. Sie liefert Strukturen mit Einzel-Eingang und Einzel-Ausgang, die Sie auf logische Genauigkeit prüfen können. Strukturierte Programmierung macht den Entwickler häufig auf Ungenauigkeiten oder mögliche Kombinationen von Eingaben aufmerksam. Strukturierte Programmierung ist kein Allheilmittel, bringt jedoch einige Ordnung in einen anderweitig verwirrenden Vorgang. Die strukturierte Programmierung sollte auch bei der Fehlersuche, beim Testen und Dokumentieren helfen.

Strukturierte Programmierung ist nicht einfach. Der Programmierer muß nicht nur die Aufgabe entsprechend definieren, er muß auch die Logik sorgfältig durcharbeiten. Dies ist mühsam und schwierig, kann jedoch ein klar geschriebenes und funktionierendes Programm ergeben.

BEGRENZER FÜR STRUKTUREN

Die speziellen Strukturen, die wir vorgestellt haben, sind nicht ideal, und sind häufig mühsam in der Anwendung. Zusätzlich kann es schwierig sein zu unterscheiden, wo eine eine Struktur endet und eine andere beginnt, speziell wenn sie verschachtelt sind. Theoretiker könnten bessere Strukturen in der Zukunft liefern, oder Entwickler könnten eigene hinzufügen. Irgend eine Art eines Abschlußzeichens für jede Struktur erscheint erforderlich, da ein Verzahnen nicht immer die Situation deutlicher macht. "End" ist ein logisches Abschlußzeichen für eine "Führe aus-solange"-Schleife. Dies ist ein deutliches Abschlußzeichen, jedoch für die "wenn-dann-sonst" (if-then-else)-Anweisung haben einige Theoretiker "endif" oder "fi" ("if" umgedreht) vorgeschlagen, aber diese beiden sind unhandlich und tragen nicht zur Lesbarkeit der Programme bei.

REGELN FÜR STRUKTURIERTE PROGRAMMIERUNG

Wir schlagen folgende Regeln für die Anwendung strukturierter Programmierung vor:

1) **Beginnen Sie mit dem Schreiben eines grundlegenden Flußdiagrammes,** um die Logik in dem Programm zu definieren.
2) **Beginnen Sie mit den Kostruktionen "wenn-dann-sonst" und "führe aus-solange".** Sie sind als kompletter Satz bekannt, das heißt, jedes Programm kann mit den Ausdrücken dieser Strukturen geschrieben werden.
3) **Bringen Sie jede Ebene in einen gewissen Abstand von der vorhergehenden Ebene an,** so daß Sie wissen, zu welcher entsprechenden Anweisung sie gehören.
4) **Verwenden Sie Abschlußzeichen für jede Struktur,** z.B. "end" für die "führe aus-solange" und "endif" oder "fi" für die "wenn-dann-sonst". Die Abschlußzeichen sowie die Trennung sollte die Programme angemessen klar machen.
5) **Achten Sie auf Einfachheit und Lesbarkeit.** Lassen Sie genügend Zwischenräume, verwenden Sie bedeutungsvolle Namen und machen Sie Ausdrücke so klar wie möglich. Versuchen Sie nicht die Logik auf Kosten der Klarheit zu minimisieren.
6) **Kommentieren Sie das Programm** ausführlich.
7) **Prüfen Sie die Logik.** Versuchen Sie alle extremen Fälle oder speziellen Bedingungen und einige Beispiele. Einen logischen Fehler, den Sie jetzt auffinden, wird Ihnen später viel Ärger ersparen.

"TOP-DOWN"-ENTWICKLUNG (SCHRITTWEISE VERFEINERUNG)

BOTTOM-UP-ENTWICKLUNG

Das verbleibende Problem besteht darin, die Module oder Strukturen zu prüfen und zusammenzusetzen. Sicher wollen wir, daß eine große Aufgabe in Unter-Aufgaben aufgeteilt wird. Aber wie prüfen wir die Unter-Aufgaben isoliert und bringen sie dann zusammen? Das Standard-Verfahren, genannt "Bottom-up"-Entwicklung, benötigt zusätzliche Arbeit für das Testen und Fehlersuchen und läßt die gesamte Integrations-Aufgabe bis zum Schluß. Was wir benötigen, ist ein Verfahren, das uns das Testen und Fehlersuchen in der tatsächlichen Programm-Umgebung ermöglicht und die System-Integration in eine Reihe von modularen Aufgaben unterteilt.

TOP-DOWN-ENTWICKLUNGS-METHODEN

PROGRAMM-STÜMPFE

ERWEITERUNG VON PROGRAMM-STÜMPFEN

VORTEILE DER TOP-DOWN-ENTWICKLUNG

Diese Verfahren ist das sogenannte Top-Down-Verfahren, das man im Deutschen am besten mit "schrittweiser Verfeinerung" bezeichnen könnte. Hier beginnen wir mit dem Schreiben eines Gesamt-Überwachungs-Programmes. Wir ersetzen die undefinierten Unterprogramme durch Programm-"Stümpfe" (stubs), Zwischenprogramme, die entweder die Eingabe aufzeichnen, die Antwort auf ein gewähltes Testprogramm liefern oder auch nichts ausführen. Wir testen dann das Überwachungsprogramm um festzustellen, ob seine Logik korrekt ist.

Wir setzen mit der Erweiterung der Stümpfe fort. Jeder Stumpf wird häufig Unter-Aufgaben enthalten, die wir zeitweilig als Stümpfe darstellen werden. Dieser Vorgang der Erweiterung, Fehlersuche und Testen wird fortgesetzt, bis alle Stümpfe durch ein funktionierendes Programm ersetzt werden. Beachten Sie, daß Testen und Zusammensetzung in jeder Ebenen auftreten, statt alle am Ende. Es sind keine speziellen Treiber oder Datenerzeugungs-Programme erforderlich. Wir bekommen eine Vorstellung, wo wir uns genau innerhalb der Entwicklung befinden. **Das Verfahren der schrittweisen Verfeinerung nimmt die modulare Programmierung an und ist ebenfalls mit strukturierter Programmierung kompatibel.**

NACHTEILE DER SCHRITTWEISEN VERFEINERUNG

Die Nachteile der schrittweisen Verfeinerung sind:

1) Die gesamte Entwicklung kann die System-Hardware wenig berücksichtigen.
2) Sie benützt nicht besonders gut die Vorteile der existierenden Software.
3) Ein geeigneter Stumpf kann schwierig zu schreiben sein, speziell wenn der gleiche Stumpf ordnungsgemäß an mehreren verschiedenen Stellen arbeiten muß.
4) Das Verfahren der schrittweisen Verfeinerung ergibt keine allgemein nützlichen Module.
5) Fehler in der obersten Ebene haben katastrophale Auswirkung, während Fehler bei der Bottom-up-Entwicklung gewöhnlich auf spezielle Module beschränkt sind.

In großen Programmier-Vorhaben hat sich das Verfahren der schrittweisen Verfeinerung als große Verbesserung für die Produktivität des Programmierers erwiesen. Jedoch die meisten dieser Projekte haben auch die Bottom-up-Entwicklung in Fällen verwendet, in denen das Verfahren der schrittweisen Verfeinerung eine Menge zusätzlicher Arbeit ergeben hätte.

Die schrittweise Verfeinerung ist ein nützliches Hilfsmittel, das jedoch nicht bis ins Extrem verwendet werden sollte. Sie besitzt die gleichen Regeln für das Testen von Systemen und der Integration, wie die strukturierte Programmierung für die Modul-Entwicklung. Das Verfahren ist jedoch allgemeiner verwendbar, da es nicht wirklich die Verwendung programmierter Logik annimmt. Die schrittweise Verfeinerung ergibt nicht immer die effizienteste Ausführung.

BEISPIELE

Reaktion auf einen Schalter

SCHRITTWEISE VERFEINERUNG DES SCHALTER- UND LAMPENSYSTEMS

Das erste Beispiel für die strukturierte Programmierung demonstriert die schrittweise Verfeinerung. Das Programm war:

```
SWITCH = OFF
do while SWITCH = OFF
  READ SWITCH
end
LIGHT = ON
DELAY 1
LIGHT OFF
```

Die meisten dieser Anwendungen sind wirklich Stümpfe, da keine von ihnen voll definiert ist. Was bedeutet zum Beispiel LIES SCHALTER? Wenn der Schalter ein Bit des Eingangs-Ports SPORT ist, bedeutet es in Wirklichkeit

```
SCHALTER = SPORT UND SMASK
```

wobei SMASK ein "1"-Bit in der entsprechenden Position besitzt.

Ähnliches bedeutet VERZÖGERUNG 1 tatsächlich (wenn der Prozessor selbst die Verzögerung liefert):

```
REG = COUNT
do while REG ≠ 0
  REG = REG - 1
end
```

COUNT (ZÄHLUNG) ist die entsprechende Zahl zur Lieferung einer Verzögerung von einer Sekunde. **Die erweiterte Version des Programmes ist:**

```
SWITCH = 0
do while SWITCH = 0
  SWITCH = SPORT AND MASK
end
LIGHT = ON
REG = COUNT
do while REG ≠ 0
  REG = REG - 1
end
LIGHT = NOT (LIGHT)
```

Sicher ist diese Programm expliziter und könnte leichter in tatsächliche Befehle oder Anweisungen übersetzt werden.

Der Speicher-Lader mit Schaltern

ENTWICKLUNG MIT SCHRITTWEISER VERFEINERUNG DES SPEICHER-LADERS MIT SCHALTERN

Dieses Beipiel ist komplexer als das erste Beispiel, so daß wir systematisch vorgehen müssen. **Hier enthält das strukturierte Programm wieder tatsächlich Stümpfe.**

Zum Beispiel, wenn die HIGH ADDRESS-Taste ein Bit des Eingangs-Ports CPORT ist, so bedeutet "wenn HIADDRBUTTON = 1" wirklich:

1) Gib von CPORT ein.
2) Komplementiere.
3) Logisch UNDiere mit HAMASK.

wobei HAMASK eine "1" in der entsprechenden Bit-Position besitzt und andernfalls Nullen. Ähnlich bedeutet die Bedingung "wenn DATABUTTOM = 1" in Wirklichkeit:

1) Gib von CPORT ein.
2) Komplementiere.
3) UNDiere logisch mit DAMASK.

Daher könnten die anfänglichen Stümpfe den Tasten Werte zuweisen, zum Beispiel:

```
HIADDRBUTTON = 0
LOADDRBUTTON = 0
DATABUTTON = 0
```

Ein Ablauf des Überwachungsprogrammes sollte zeigen, daß es den hierin enthaltenen "dann (else)"-Weg durch die "if-then-else"-Strukturen nimmt und die Schalter niemals liest. Ähnlich, wenn der Stumpf

```
HIADDRBUTTON = 1
```

wäre, sollte das Überwachungsprogramm in der Schleife "do while HIADDRBUTTON = 1"warten, bis die Taste losgelassen wird. Dieser einfache Durchlauf prüft die gesamte Logik.

Nun können wir mit der Erweiterung jedes Stumpfes beginnen und sehen, ob die Erweiterung ein vernünftiges Gesamtresultat ergibt. Beachten Sie, wie Fehlersuche und Testen in einer einfachen und modularen Weise fortschreitet. Wir erweitern den Stumpf HIADDRBUTTON = 1 auf

```
LIES CPORT
HIADDRBUTTON = NICHT (CPORT) UND HAMASK
```

Dieses Programm sollte warten, bis die Taste HIGH ADDRESS geschlossen ist. Das Programm sollte dann die Werte der Schalter auf den Lampen anzeigen. Dieser Durchlauf prüft die richtige Reaktion auf die HIGH ADDRESS-Taste.

Wir erweitern dann das Tastenmodul LOW ADDRESS auf

```
LIES CPORT
HIADDRBUTTON = NICHT (CPORT) UND LAMASK
```

Mit der LOW ADDRESS-Taste in geschlossener Lage sollte das Programm die Werte der Schalter auf den Lampen anzeigen. Dieser Durchlauf prüft die richtige Reaktion auf die LOW ADDRESS-Taste.

Ähnlich können wir das Modul DATA-Taste erweitern und die richtige Reaktion auf diese Taste prüfen. Das gesamte Programm wurde dann auf systematische Weise geprüft.

Wenn alle Stümpfe erweitert wurden, so sind die Codier-, Fehlersuch- und Teststufen abgeschlossen. Natürlich müssen wir genau wissen, welche Ergebnisse jeder Stumpf erzeugen sollte. Zahlreiche logische Fehler werden jedoch bei jeder Ebene erkennbar werden, ohne irgend eine weitere Erweiterung.

Das Verifikations-Terminal

SCHRITTWEISE VERFEINERUNG DES VERIFIKATIONS-TERMINALS

Dieses Beispiel besitzt natürlich mehrere Detail-Ebenen. **Wir könnten mit dem folgenden Programm beginnen** (siehe Bild 13-17 für ein Flußdiagramm):

```
KEYBOARD
ACK = 0
do while ACK = 0
  TRANSMIT
  RECEIVE
end
DISPLAY
```

Hier sind TASTATUR, SENDE, EMPFANGE und ANZEIGE Programmstümpfe, die später erweitert werden. TASTATUR zum Beispiel könnte einfach eine überprüfte zehnstellige Zahl in das entsprechende Puffer plazieren.

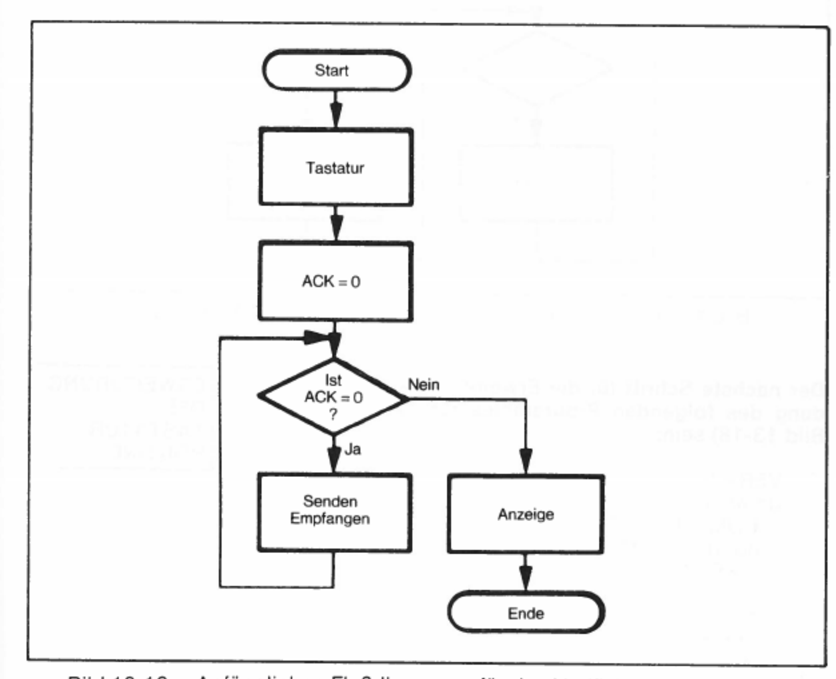


Bild 13-19. Anfängliches Flußdiagramm für das Verifizierungs-Terminal.

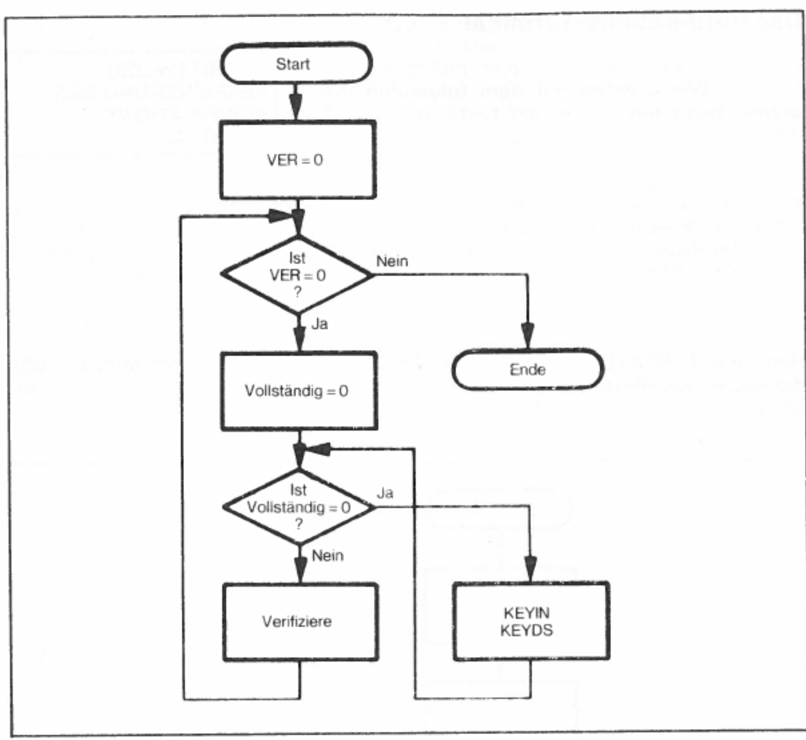


Bild 13-20. Flußdiagramm für die erweiterte TASTATUR-Routine.

ERWEITERUNG DER TASTATUR-ROUTINE

Der nächste Schritt für die Erweiterung könnte in der Bildung des folgenden Programmes für TASTATUR (siehe Bild 13-18) sein:

```
VER = 0
do while VER = 0
  COMPLETE = 0
  do while COMPLETE = 0
    KEYIN
    KEYDS
  end
  VERIFY
end
```

Hier bedeutet VER = 0, daß eine Eingabe nicht verifiziert wurde. VERVOLLSTÄNDIGE (COMPLETE) = 0 bedeutet, daß diese Eingabe unvollständig ist. KEYIN und KEYDS sind die Tastatur-Eingabe- und Anzeige-Routinen. VERIFY prüft die Eingabe. Ein möglicher Stumpf für KEYIN würde einfach eine zufällige Eingabe (von einer Zufallszahlen-Tabelle- oder Generator) in den Puffer plazieren und COMPLETE auf 1 setzen.

Wir würden mit einer ähnlichen Erweiterung, Fehlersuche und Testen von SENDEN, EMPFANG und ANZEIGE fortsetzen. Beachten Sie, daß Sie jedes Programm um eine Ebene erweitern sollten, so daß Sie nicht die Zusammensetzung des ganzen Programms auf einmal ausführen. Sie müssen selbst über die Definition der Ebene entscheiden. Zu kleine Schritte verschwenden Zeit, während ein zu großer Schritt Sie zu den Problemen der System-Integration zurückwirft, die die Entwicklung mit der schrittweisen Verfeinerung lösen sollte.

ÜBERBLICK ÜBER DAS VERFAHREN DER SCHRITTWEISEN VERFEINERUNG

Das Verfahren der schrittweisen Verfeinerung bringt Ordnung in die Test- und Integrations-Phasen der Programm-Entwicklung. Sie liefert ein systematisches Verfahren zur Erweiterung eines Flußdiagrammes oder der Aufgaben-Definition auf eine Ebene, die zum tatsächlichen Schreiben eines Programmes erforderlich ist. Zusammen mit strukturierter Programmierung bilden sie ein vollständiges Entwicklungs-Verfahren.

Wie die strukturierte Programmierung, ist auch das Verfahren mit der schrittweisen Verfeinerung nicht einfach. Der Entwickler muß die Aufgabe sorgfältig definiert haben und sich systematisch durch die Ebenen arbeiten. Hier erscheint die Methode wieder mühsam, aber der Vorteil kann erheblich sein, wenn der Entwickler tatsächlich diesen Regeln folgt.

FORMAT DER TOP-DOWN-ENTWICKLUNG

Wir empfehlen die folgende Lösung für das Verfahren der schrittweisen Verfeinerung:

1) **Beginnen Sie mit einem grundlegenden Flußdiagramm.**
2) **Machen Sie die Stümpfe so vollständig und so getrennt wie möglich.**
3) **Definieren Sie genau alle Ergebnisse jedes Stumpfes** und wählen Sie die entsprechenden Test-Verfahren.
4) **Prüfen Sie jede Ebene sorgfältig** und systematisch.
5) **Verwenden Sie die Strukturen der strukturierten Programmierung.**
6) **Erweitern Sie jeden Stumpf um eine Ebene.** Versuchen Sie nicht zu viel in einem Schritt auszuführen.
7) **Achten Sie sorgfältig auf gemeinsame Aufgaben und Datenstrukturen.**
8) **Testen Sie und suchen Sie die Fehler nach jeder Struktur-Erweiterung.** Versuchen Sie nicht eine ganze Ebene auf einmal auszuführen.
9) **Achten Sie darauf, was die Hardware ausführen kann. Scheuen Sie sich nicht zu stoppen und auch eine kleine "Bottom-up"-Entwicklung auszuführen, wenn diese erforderlich ist.**

ÜBERBLICK ÜBER DIE AUFGABEN-DEFINITION UND PROGRAMM-ENTWICKLUNG

Sie sollten beachten, daß wir ein ganzes Kapitel aufwendeten, ohne irgendeinen speziellen Mikroprozessor oder Assemblersprache erwähnt zu haben, und ohne das Schreiben einer einzigen Zeile eines tatsächlichen Codes. Hoffentlich wissen Sie nun mehr über die Beispiele, als wenn wir von Ihnen am Anfang verlangt hätten, Programme zu schreiben. Obwohl wir häufig glauben, daß das Schreiben von Computerbefehlen der wesentliche Teil der Software-Entwicklung ist, ist es nur eine der leichtesten Stufen.

Sobald Sie einige Programme geschrieben haben, wird die Stufe des Codierens sehr einfach werden. Sie werden bald den Befehlssatz lernen, erkennen welche Befehle wirklich nützlich sind und werden dann finden, daß andere Stufen der Software-Entwicklung schwierig bleiben und weniger klare Regeln besitzen.

Wir haben hier einige Wege vorgeschlagen, um diese wichtigen frühen Stufen in ein System zu bringen. In der Stufe der Aufgaben-Definition mußten Sie alle Eigenschaften des Systems definieren, seine Eingangs- und Ausgangs-Signale, Verarbeitung, Zeit- und Speicher-Beschränkungen und die Handhabung von Fehlern. Sie mußten besonders überlegen, wie das System mit größeren Systemen zusammenarbeitet, von dem es ein Teil ist und ob das größere System elektrische Geräte, mechanische Geräte oder einen Menschen als Bedienenden umfaßt. Sie müssen in dieser Stufe mit den Überlegungen beginnen, wie Sie das System leicht zu bedienen und zu warten machen.

In der Stufe der Programm-Entwicklung können Ihnen verschiedene Techniken helfen, systematisch die Logik Ihres Programmes zu spezifizieren und zu dokumentieren. Modulare Programmierung zwingt Sie zur Aufteilung des gesamten Programmes in kleine abgegrenzte Module. Strukturierte Programmierung ermöglicht einen systematischen Weg zur Definition der Logik derartiger Module, während die Entwicklung mit schrittweiser Verfeinerung ein systematisches Verfahren zur Zusammensetzung und Testen dieser ist. Natürlich kann Sie niemand zwingen, alle diese Verfahren zu verwenden. Sie sind in der Tat mehr als Richtlinien gedacht. Aber sie ermöglichen eine einheitliche Lösung für die Definitions- und Entwicklungs-Stufen und Sie sollten Sie als eine Basis für die Entwicklung Ihrer eigenen Lösung betrachten.

LITERATUR

1) Ballard, D. R., "Designin Fail-Safe Micorpocessor Systems," Electronics, January 4, 1979, pp. 139-143.

"A Desinger's Guide to Signature Analysis," Hewlett-Packard Application Note 222, Hewlett-Packard, Inc., Palo Alto, CA, 1977.

Donn, E. S. and M. D. Lippman, "Efficient and Effective Microcomputer Testing Requires Careful Preplanning," EDN, February 20, 1979, pp. 97-107 (includes self-test examples for 6502).

Gordon, G. and H. Nadig, "Hexadecimal Signatures Identify Troublespots in Microprocessor Systems," Electronics, March 3, 1977, pp. 89-96.

Neil, M. and R. Goodner, "Designing a Serviceman's Needs into Microprocessor-Based Systems," Electronics, March 1, 1979, pp. 122-128.

Schweber, W. and L. Pearce, "Software Signature Analysis Identifies and Checks PROM's," EDN, November 5, 1978, pp. 79-81.

Srini, V. P., "Fault Diagnosis of Microprosessor Systems," Computer, January 1977, pp. 60-65.

2 For a brief discussion of human factors considerations, see G. Morris, "Make Your Next Instrument Design Emphasizes User Needes and Wants," EDN, October 20, 1978, pp. 100-105.

3) D. L. Parnas (see the references below) has been a leader in the area of modular programming.

4) Collected by B. W. Unger (see reference below).

5) Formulated by D. L. Parnas.

6) F. Heubach, "Strukturierte Programmierung auch bei Mikrocomputern", Elektronik 1977, Heft 10, Seite 113 bis 118.

Zu den folgenden Literaturstellen finden Sie weitere Informationen über die Aufgaben-Definition und Programm-Entwicklung:

Chapin, N., Flowcharts, Auerbach, Princeton, N. J., 1971.

Dalton, W. F., "Design Microcomputer Software like Other Systems–Systematically," Electronics, January 19, 1978, pp. 97-101.

Dijkstra, E. W., A Discipline of Programming, Prentice-Hall, Englewood Cliffs, N. J., 1976.

Halstead, M. H., Elements of Software Science, American Elsevier, New York, 1977.

Hughes, J. K. and J. I. Michtom, A Structured Approach to Programming, Prentice-Hall, Englewood Cliffs, N. J., 1977.

Morgan, D. E. and D. J. Taylor, "A Survey of Methods for Achieving Reliable Software," Computer, February 1977, pp. 44-52.

Myers, W., "The Need for Software Engineering," Computer, February 1978, pp. 12-25.

Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, December 1972, pp. 1053-1058.

Parnas, D. L., "A Technique for the Specification of Software Modules with Examples," Communications of the ACM, May 1973, pp. 330-336.

Phister, M. Jr., Data Processing Technology and Economics, Santa Monica Publishing Co., Santa Monica, CA, 1976.

Schneider, V., "Prediction of Software Effort and Project Duration–Four New Formulas," SIGPLAN Notices, June 1978, pp. 49-59.

Schneiderman, B., et al., "Experimental Investignations of the Utility of Detailed Flowcharts in Programming," Communications of the ACM, June 1977, pp. 373-381.

Tausworthe, R. C., Standardized Development of Computer Software, Prentice-Hall, Englewood Cliffs, N. J., 1977.

Unger, B. W., "Programming Languages for Computer System Simulation," Simulation, April 1978, pp. 101-110.

Wirth, N., Algorithms + Data Structures = Programs, Prentice-Hall, Englewood Cliffs, N. J., 1976.

Wirth, N., Systematic Programming; an Introduction, Prentice-Hall, Englewood Cliffs, N. J., 1973.

Yourdon, E. U., Techniques of Program Structure and Design, Prentice-Hall, Englewood Cliffs, N. J., 1975.

Kapitel 14
FEHLERSUCHE UND TESTEN

Wie wir am Beginn des vorhergehenden Kapitels festgestellt haben, gehören Fehlersuche und Testen zu den zeitraubendsten Stufen der Software-Entwicklung. **Obwohl Verfahren wie modulare Programmierung, strukturierte Programmierung und Entwicklung mit schrittweiser Verfeinerung die Programme vereinfachen und die Häufigkeit von Fehlern verringern können, sind Fehlersuche und Testen noch immer schwierig,** da sie so schlecht zu definieren sind. Die Auswahl einer ausreichenden Anzahl von Testdaten ist ebenfalls selten ein klarer oder wissenschaftlicher Vorgang. Das Finden von Fehlern erscheint häufig als reines Glücksspiel. Es gibt wenige Aufgaben, die manchmal so frustrierend sein können, wie das Suchen von Fehlern in Programmen.

Dieses Kapitel wird zuerst die verfügbaren Hilfsmittel bei der Fehlersuche beschreiben. Es werden dann grundlegende Fehlersuch-Verfahren besprochen, die häufigsten Arten von Fehlern beschrieben und einige Beispiele für Fehlersuche in Programmen vorgestellt. Im letzten Abschnitt wird beschrieben, wie Testdaten und Testprogramme auszuwählen sind.

Wir können nicht mehr tun, als den Zweck der meisten Fehlersuch-Hilfsmittel zu beschreiben. In diesem Bereich gibt es kaum eine Standardisierung und es ist nicht möglich, alle momentan verfügbaren Bausteine und Programme zu besprechen. Die Beispiele sollten Ihnen eine Vorstellung von der Anwendung, den Vorteilen und den Grenzen spezieller Hardware- oder Software-Hilfsmittel geben.

EINFACHE FEHLERSUCH-HILFSMITTEL

Die einfachsten Hilfsmittel für das Aufsuchen von Fehlern sind:

- **Eine Einzelschritt-Einrichtung**
- **Eine Haltepunkt-Einrichtung**
- **Ein Programm für einen Register-Auszug**
- **Ein Programm für einen Speicher Auszug**

EINZEL-SCHRITT

Die Einzelschritt-Einrichtung gestattet Ihnen, das Programm schrittweise, zu durchlaufen.
Die meisten auf dem 6502 basierende Mikrocomputer besitzen diese Möglichkeit, da die Schaltung verhältnismäßig einfach ist. **Natürlich sind die einzigen sichtbaren Dinge, wenn der Computer einen Einzelschritt ausführt, die Zustände der von Ihnen überwachten Ausgangsleitungen.** Die wichtigsten Leitungen sind:

- Datenbus
- Adressenbus
- Steuerleitungen
- SYNC (Synchronisation) und READ/$\overline{\text{WRITE}}$

Wenn Sie diese Leitungen (entweder in Hardware oder in Software) überwachen, so werden Sie die jeweiligen Adressen, Befehle und Daten bei Ausführung des Programmes sehen können. Sie werden imstande sein zu sagen, welche Art von Operationen die CPU ausführt. Diese Informationen werden Sie über Fehler, wie falsche Sprungbefehle, ausgelassene oder falsche Adressen, fehlerhafte Operationscodes oder falsche Datenwerte unterrichten. Sie können jedoch nicht den Inhalt der Register und Flags erkennen, ohne einige zusätzliche Fehlersuch-Möglichkeiten oder ohne einer speziellen Sequenz von Befehlen. Zahlreiche Operationen des Programmes können nicht unmittelbar geprüft werden.

GRENZEN DES EINZELSCHRITT-VERFAHRENS

Es gibt zahlreiche Fehler, die Sie mit dem Einzelschritt-Verfahren nicht auffinden können. Dies sind Fehler in der zeitlichen Steuerung und Fehler in den Unterbrechungs- oder DMA-Systemen. Ferner ist das Einzelschritt-Verfahren sehr langsam, wobei ein Programm mit einer Geschwindigkeit ausgeführt wird, die etwa ein Millionstel der Geschwindigkeit des Prozessors beträgt. Wenn Sie daher nur eine Sekunde der echten Prozessorzeit in Einzelschritten durchlaufen, würde dies mehr als zehn Tage dauern. Das Einzelschritt-Verfahren ist nur dann nützlich, wenn kurze Befehls-Sequenzen zu überprüfen sind.

HALTEPUNKT

Ein Haltepunkt (breakpoint) ist eine Stelle, bei der das Programm automatisch halten oder warten wird, so daß der Anwender den momentanen Status des Systems prüfen kann. Das Programm wird gewöhnlich nicht neuerlich starten, bis der Operator dies wünscht. Der Haltepunkt gestattet Ihnen, einen ganzen Abschnitt eines Programmes zu überprüfen. Wenn Sie daher feststellen wollen, ob eine Initialisierungs-Routine korrekt ist, können Sie einen Haltepunkt am Ende dieser plazieren und das Programm ablaufen lassen. Sie können dann Speicherplätze und Register prüfen um festzustellen, ob der gesamte Abschnitt richtig arbeitet. Beachten Sie jedoch, daß falls der Abschnitt nicht in Ordnung ist, Sie trotzdem den Fehler noch aufsuchen müssen, entweder mit früheren Haltepunkten oder mit dem Einzelschritt-Verfahren.

Haltepunkte ergänzen das Einzelschritt-Verfahren. Sie können Haltepunkte verwenden, entweder um den Fehler zu lokalisieren oder die Abschnitte zu durchlaufen von denen Sie wissen, daß sie korrekt sind. Dann können Sie die detaillierte Fehlersuche mit dem Einzelschritt-Verfahren ausführen. Beachten Sie, daß Haltepunkte die zeitliche Steuerung des Programmes nicht beeinflussen, sie können daher zum Prüfen von Eingabe/Ausgabe und Unterbrechungen eingesetzt werden.

BRK ALS HALTEPUNKT

Haltepunkte verwenden häufig einen Teil oder das gesamte Mikroprozessor-Unterbrechungssystem.

Einige Mikroprozessoren besitzen eine spezielle Software-Unterbrechung oder "Fallen" (Traps), die als Haltepunkte verwendet werden können. Der Befehl BRK (Force Break) des 6502 kann als Haltepunkt dienen. Wenn Sie nicht bereits die maskierbare Unterbrechung (IRQ) und die nicht-maskierbare Unterbrechung (NMI) in Ihrem Programm verwenden, so lassen sich diese Vektoren als extern gesteuerte Haltepunkte einsetzen. Tabelle 14-1 zeigt die Adressen der verschiedenen Unterbrechungs-Vektoren des 6502. Kapitel 12 beschreibt die Vektoren detaillierter. Die Haltepunkt-Routine kann den Inhalt von Registern und Speichern ausdrucken oder einfach warten (durch Ausführung eines bedingten Sprunges, abhängig von einer Schalter-Eingabe), bis der Anwender dem Computer gestattet, das Programm fortzusetzen. Aber erinnern Sie sich daran, daß die Unterbrechungen (einschließlich BRK) den Stapel und den Stapelzeiger verwenden, um die Rückkehr-Adresse und den Register-Inhalt zu speichern. Bild 14-1 zeigt eine Routine, in der BRK in einer endlosen Schleife resultiert. Der Programmierer hätte diesen Haltepunkt mit einem RESET-oder Unterbrechungs-Signal zu löschen.

Tabelle 14-1. 6502-Unterbrechungs-Vektoren.

Eingang	Vektor-Adresse (hexadezimal)
NMI	FFFA, FFFB
RESET	FFFC, FFFD
IRQ oder BRK	FFFE, FFFF

```
*=BREAK          ;ADRESSE FÜR BREAK-ROUTINE
JMP    BREAK     ;WARTE AUF DER STELLE
```

Die Unterbrechungs-Service-Routine muß einen Sprung zur Adresse BREAK erzwingen, wenn sie das Break-Kommando-Flag gesetzt findet (dies unterscheidet zwischen BRK und IRQ-Eingängen).

Bild 14-1. Eine einfache Haltepunkt-Routine.

Das einfachste Verfahren zum Einsetzen von Haltepunkten besteht im Ersetzen der ersten Bytes des Befehls durch einen BRK-Befehl, oder durch Ersetzen eines Befehls mit einem JMP- oder JSR-Befehl. Der BRK-Befehl ist vorzuziehen, da nur ein einziges Byte ersetzt werden muß, und der Haltepunkt nicht darauf folgende Befehle überschreiben wird.

EINSETZEN VON HALTEPUNKTEN

Zahlreiche Monitore besitzen die Möglichkeit des Einsetzens und Entfernens von Haltepunkten über irgendeine Art von Sprungbefehl.
Derartige Haltepunkte beeinflussen nicht die zeitliche Steuerung des Programmes, bis der Haltepunkt ausgeführt wird. Beachten Sie jedoch, daß dieses Verfahren nicht arbeiten wird, wenn ein Teil oder das gesamte Programm sich in einem ROM oder PROM befindet. Andere Monitore führen Haltepunkte aus, indem sie tatsächlich die Adressenleitungen oder den Befehlszähler in Hardware oder Software prüfen. Dieses Verfahren gestattet Haltepunkte an Adressen im ROM oder PROM, kann jedoch die zeitliche Steuerung beeinlussen, wenn die Adresse in Software geprüft werden muß. Eine leistungsfähige Möglichkeit würde dem Anwender die Eingabe einer Adresse gestatten, zu der der Prozessor die Steuerung überträgt. Eine ander Möglichkeit bestünde in einer Rückkehr, abhängig von einem Schalter:

```
        *=BRKPT            ;ADRESSE FÜR HALTEPUNKT-ROUTINE
        BIT     VIAORA     ;WARTE AUF GESCHLOSSENEN SCHALTER
        BPL     BRKPT
        RTI
```

Natürlich könnten andere VIA-Daten oder Steuerleitungen verwendet werden. Erinnern Sie sich daran, daß RTI automatisch das Statusregister wiederherstellt und die Unterbrechung wieder freigibt. Wenn die Unterbrechung über eine VIA-Steuerleitung kommt, müßte die Routine auch das zugehörige Bit im Unterbrechunsflag-Register löschen.

REGISTER-AUSZUG

Die Möglichkeit für einen Register-Auszug in einem Mikrocomputer ist ein Programm, das den Inhalt aller CPU-Register auflistet. Diese Information ist gewöhnlich nicht direkt erhältlich. **Die folgende Routine wird den Inhalt aller Register ausdrucken,** wenn wir annehmen, daß PRTHEX den Inhalt des Akkumulators in Form zweier hexadezimaler Ziffern ausdruckt. Bild 14-2 ist ein Flußdiagramm des Programmes und Bild 14-3 zeigt ein typisches Ergebnis. Wir nehmen an, daß die Routine die mit einem Befehl JUMP TO SUBROUTINE eingegeben wurde, die den alten Befehlszähler in die Spitze des Stapels speichert. Eine Unterbrechung oder ein BRK-Befehl wird sowohl den Befehlszähler wie das Statusregister in die Spitze des Stapels speichern.


```
;
;PLAZIERE ALLE CPU-REGISTER IN DEN STAPEL (PC BEREITS IM STAPEL)
;
        PHP                ;BEWAHRE STATUS AUF, FALLS ERFORDER-
                           ;  LICH (NICHT NACH IRQ)
        PHA                ;BEWAHRE INHALT DES AKKUMULATORS
                           ;  AUF
        TXA                ;BEWAHRE INDEXREGISTER X AUF
        PHA
        TYA                ;BEWAHRE INDEXREGISTER Y AUF
        PHA
        TSX                ;BEWAHRE URSPRÜNGLICHEN STAPELZEI-
                           ;  GER AUF
        TXA
        CLC
        ADC     #6         ;VERSETZE ZURÜCK ZUM URSPRÜNGLI-
                           ;  CHEN WERT
;
;DRUCKE INHALT DER REGISTER AUS
;REIHENFOLGE IST S, Y, X, A, P, PC, (LOW), PC (HIGH)
        LDY     #7         ;ANZAHL DER BYTES = 7
PRNT    LDA     $0100,X    ;HOLE EIN BYTE VOM STAPEL
        JSR     PRTHEX     ;UND DRUCKE AUS
        INX
        DEY
        BNE     PRNT
;
;SPEICHERE REGISTER VOM STAPEL ZURÜCK
;
        PLA                ;HOLE UND LÖSCHE STAPELZEIGER
        PLA                ;SPEICHERE INDEXREGISTER Y ZURÜCK
        TAY
        PLA                ;SPEICHERE INDEXREGISTER X ZURÜCK
        TAX
        PLA                ;SPEICHERE INHALT DES AKKUMULATORS
                           ;  ZURÜCK
        PLP                ;SPEICHERE STATUSREGISTER FALLS
                           ;  ERFORDERLICH ZURÜCK
        RTS                ;SPEICHERE PC UND SP ZURÜCK
```

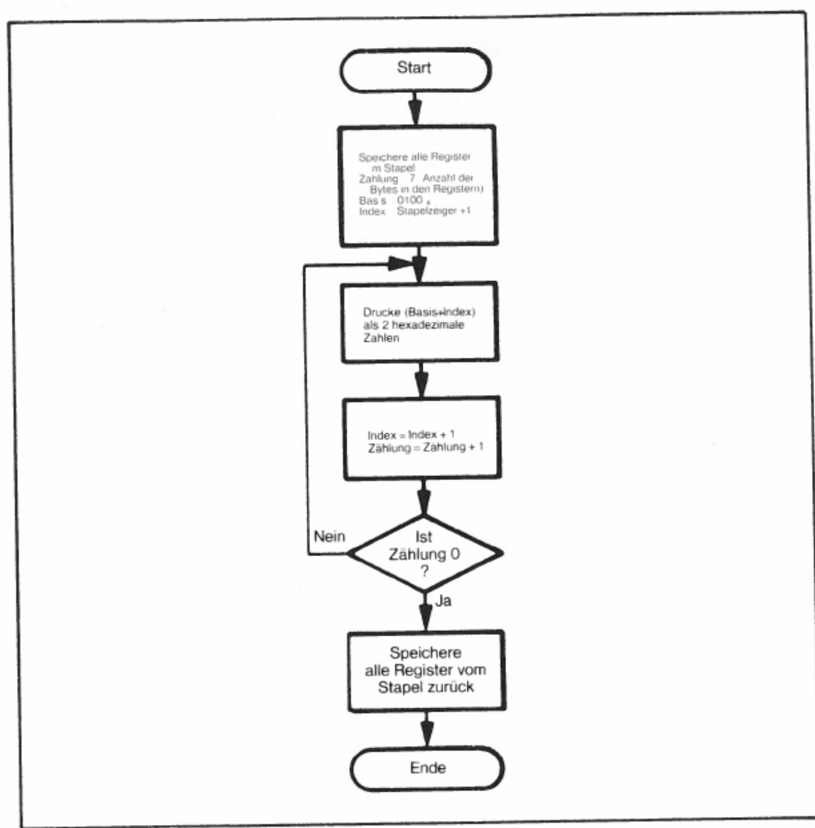


Bild 14-2. Flußdiagramm des Registerauszugs-Programm.

3E	BEFEHLSZÄHLER
FC	
86	(PSW)
1D	(A)
07	(B)
3E	(C)
23	(D)
01	(E)
17	(H)
F0	(L)
FC	STAPELZEIGER
3F	

Bild 14-3. Ergebnisse eines typischen Register-Auszugs

SPEICHER-AUSZUG

Ein Speicher-Auszug ist ein Programm, das den Inhalt des Speichers auf einem Ausgabegerät (wie etwa einem Drucker) auflistet. Dies ist ein wesentlich effizienterer Weg zur Prüfung von Daten-Anordnungen oder des gesamten Programmes, als nur das Untersuchen einzelner Stellen. Es sind jedoch sehr große Speicher-Auszüge infolge ihrer großen Menge von Informationen nicht sehr nützlich (außer um eine Menge Schmierzettel zu liefern). Auf einem langsamen Drucker kann die Ausführung auch verhältnismäßig lange dauern. **Kleine Speicher-Auszüge können jedoch dem Programmierer eine vernünftige Menge von Informationen liefern, die er noch ohne weiteres prüfen kann. Beziehungen, wie regelmäßige Wiederholungen von Datenmustern oder Versetzungen ganzer Anordnungen werden hierbei erkenntlich.**

Ein allgemeiner Auszug ist häufig verhältnismäßig schwierig zu schreiben. Der Programmierer sollte sorgfältig auf folgende Situationen achten:

1) Die Größe des Speicherbereichs überschreitet 256 Bytes, so daß ein einzelner 8-Bit-Zähler nicht ausreichend sein wird.
2) Der Abschluß-Speicherplatz ist eine Adresse, die kleiner ist als der Start-Speicherplatz. Dies sollte als Fehler behandelt werden, da der Anwender selten wünschen wird, den gesamten Speicher-Platz in einer ungewöhnlichen Reihenfolge auszudrucken.

Da die Geschwindigkeit des Speicher-Auszug gewöhnlich abhängig von der Geschwindigkeit des Ausgabegerätes ist, ist die Effizienz der Routine selten von Bedeutung. **Das folgende Programm wird Fälle ignorieren, bei denen die Start Adresse größer als die End-Adresse ist und wird Blöcke beliebiger Länge handhaben.**

Wir nehmen an, daß sich die Startadresse in den Speicher-Adressen START und START+1 befindet, sowie die Endadresse in den Speicher-Adressen LAST und LAST+1. Wir haben angenommen, daß die Adressen START und START+1 sich auf der Nullseite befinden, so daß deren Inhalt indirekt verwendet werden kann.

```
;
;DRUCKE INHALT DER SPEZIFIZIERTEN SPEICHERPLÄTZE AUS
;
DUMP    LDY   #0           ;HALTE OFFSET IMMER AUF NULL
DBYTE   LDA   LAST         ;SIND WIR OBERHALB DER ENDADRESSE?
        CMP   START
        LDA   LAST+1
        SBC   START+1
        BCC   DONE         ;JA, AUSZUG BEENDET
        LDA   (START),Y    ;NEIN, HOLE DEN INHALT DES NÄCHSTEN
                           ;  SPEICHERPLATZES
        JSR   PRTHEX       ;DRUCKE INHALT ALS 2 HEXZIFFERN AUS
        INC   START        ;INKREMENTIERE SPEICHERZEIGER
        BNE   DBYTE
        INC   START+1
        JMP   DBYTE
DONE    RTS
```

Es gibt keinen direkten Weg, um den 16-Bit-Vergleich und Inkrementierung auszuführen, das durch diese Routine erforderlich ist.

Bild 14-4 zeigt den Ausdruck eines Auszugs der Speicherplätze 1000 bis 101F.

```
23  1F  60  54  37  28  3E  00
6E  42  38  17  59  44  98  37
47  36  23  81  E1  FF  FF  5A
34  ED  BC  AF  FE  FF  27  02
```

Bild 14-4. Ergebnisse eines typischen Speicher-Auszugs

Diese Routine arbeitet ordnungsgemäß in dem Fall, bei dem Start- und End-Speicherplätze gleich sind. (Versuchen Sie es!) Sie werden die Ergebnisse sorgfältig zu interpretieren haben, wenn der Bereich des Auszugs den Stapel beinhaltet, da das Auszugs-Unterprogramm selbst den Stapel verwendet. PRTHEX kann auch Speicher- und Stapel-Plätze ändern.

Bei einem Speicherauszug können die Daten in einer Vielzahl unterschiedlicher Arten dargestellt werden. Gebräuchliche Formen sind ASCII-Zeichen oder Paare von hexadezimalen Ziffern für 8-Bit-Werte und vier hexadezimale Ziffern für 16-Bit-Werte. Das Format sollte entsprechend der vorgesehenen Verwendung des Auszuges gewählt werden. Es ist nahezu immer einfacher einen Objektcode-Auszug zu interpretieren, wenn er in hexadezimaler Form anstatt in ASCII-Form dargestellt wird.

Ein gebräuchliches und nützliches Auszugs-Format ist hier dargestellt:

```
1000    54 68 65 20 64 75 6D 70    Der Auszug
```

Jede Zeile besteht aus drei Teilen. Die Zeile beginnt mit der hexadezimalen Adresse der ersten Bytes, das auf der Zeile dargestellt wird. Der Adresse folgen acht oder 16 Bytes in hexadezimaler Form. Zuletzt erfolgt die ASCII-Darstellung der gleichen acht oder 16 Bytes. Versuchen Sie das Speicherauszugs-Programm so neu zu schreiben, daß es die Adressen und die ASCII-Zeichen sowie die hexadezimale Form der Speicherinhaltes druckt.

FORTSCHRITTLICHERE FEHLERSUCH-HILFSMITTEL

Die fortschrittlichsten Hilfsmittel für die Fehlersuche, die am meisten verwendet werden sind:

- **Simulatorprogramme zum Prüfen der Software**
- **Logik-Analysatoren zum Prüfen der Signale und der zeitlichen Steuerung**

Von diesen beiden Hilfsmitteln existieren zahlreiche Varianten und wir werden nur die Standard-Eigenschaften besprechen.

SOFTWARE-SIMULATOR

Der Simulator stellt das Computer-Äquivalent des "Blei-stift- und Papier-Verfahrens" dar. Es handelt sich um ein Computerprogramm, das den Arbeitszyklus eines anderen Computers durchläuft, wobei es den Inhalt aller Register, Flags und Speicherplätze verfolgt. Wir könnten dies natürlich manuell ausführen, was jedoch sehr mühsam wäre und eine sorgfältige Beobachtung des Einflusses jedes Befehls voraussetzen würde. Das Simulatorprogramm dagegen wird niemals müde oder verwirrt, und vergißt keinen Befehl oder Register.

Die meisten Simulatoren sind große FORTRAN-Programme. Sie können gekauft oder auf Time-Sharing-Diensten verwendet werden. Der 6502-Simulator ist in zahlreichen Versionen von verschiedenen Lieferanten erhältlich.

Typische Simulator-Eigenschaften sind:

1) **Eine Haltepunkt-Möglichkeit.** Gewöhnlich können Haltepunkte gesetzt werden, nachdem eine bestimmte Anzahl von Zyklen ausgeführt wurde, wenn auf einen Speicherplatz oder einen Satz von Speicherplätzen Bezug genommen wurde, wenn der Inhalt eines Speicherplatzes oder einer Anzahl von Speicherplätzen geändert wurde oder bei anderen Bedingungen.
2) **Die Möglichkeit für Register- und Speicher-Auszüge,** die Werte der Speicherplätze, Register und E/A-Ports anzeigen.
3) **Eine Ablaufverfolgung (trace),** die den Inhalt spezieller Register oder Speicherplätze ausdruckt, wann immer das Programm diese ändert oder verwendet.
4) **Eine Lade-Möglichkeit,** die Ihnen das Setzen von Anfangswerten oder deren Änderung während der Simulation gestattet.

Einige Simulatoren können auch Eingaben/Ausgaben, Unterbrechungen und sogar DMA simulieren.

Das Simulator-Programm besitzt zahlreiche Vorteile:

1) Es kann eine vollständige Beschreibung des Computer-Status liefern, da das Simulator-Programm nicht durch Anschlußprobleme oder andere physikalische Probleme eingeschränkt ist.
2) Es kann Haltepunkte, Auszüge, Tracer (Ablaufverfolger) und andere Möglichkeiten liefern, ohne iregendeinen Teil des Speichers oder Steuersystems des Prozessors zu verwenden. Diese Möglichkeiten werden daher das Anwenderprogramm nicht stören.
3) Programme, Startpunkte und andere Bedingungen sind leicht zu ändern.

4) Alle Möglichkeiten eines großen Computers, einschließlich peripherer Geräte und Software sind für den Mikroprozessor-Entwickler verfügbar.

Andererseits ist das Simulator-Programm durch seine Software-Grundlage und seine Trennung vom realen Mikrocomputer begrenzt. Die wesentlichsten Grenzen sind:

1) Das Simulator-Programm kann nicht bei zeitlichen Steuerproblemen helfen, da es wesentlich langsamer als in Echtzeit arbeitet und keine tatsächliche Hardware oder Interfaces nachbildet.
2) Das Simulator-Programm kann das Eingabe/Ausgabe-System nicht voll ersetzen.
3) Das Simulator-Programm ist gewöhnlich verhältnismäßig langsam. Die Wiedergabe einer Sekunde echter Prozessorzeit kann Stunden von Computerzeit erfordern. Die Verwendung des Simulators kann ziemlich aufwendig werden.

Das Simulator-Programm stellt die Software-Seite der Fehlersuche dar. Es besitzt die typischen Vorteile und Grenzen der gesamten Software-Lösung. Das Simulator-Programm kann einen Einblick in die Programmlogik und ander Software-Probleme geben, kann jedoch nicht bei der zeitlichen Steuerung, E/A- und anderen Hardware-Problemen helfen.

LOGIG-ANALYSATOR

Der Logik- oder Mikroprozessor-Analysator ist die Hardware-Lösung für die Fehlersuche. Grundlegend ist der Analysator die parallele digitale Version des Standard-Oszillografen. Der Analysator stellt Informationen in binär, hexadezimal oder Mnemonik-Form auf einer Kathodenstrahlröhre dar, und besitzt eine Vielzahl von Trigger-Möglichkeiten, Schwellwert-Einstellungen und Eingängen. Die meisten Analysatoren besitzen einen Speicher, so daß sie auch den vorhergehenden Inhalt der Busse darstellen können.

Das Standard-Verfahren besteht im Einstellen der Triggerung auf ein bestimmtes Ereignis, etwa dem Auftreten einer bestimmten Adresse auf dem Adressenbus oder eines Befehls auf dem Datenbus. Man könnte zum Beispiel den Analysator triggern, wenn der Mikrocomputer versucht, Daten in eine bestimmte Adresse zu speichern oder einen Eingabe- oder Ausgabebefehl ausführt. Man kann sich dann die Folge der Vorgänge ansehen, die vor dem Haltepunkt auftreten. **Übliche Probleme, die man auf diese Weise finden kann, sind kurze Spannungsspitzen (glitches), falsche Signal-Sequenzen, überlappende Impulse und andere Fehler in den zeitlichen Steuersignalen. Natürlich könnte ein Software-Simulator nicht zur Feststellung derartiger Fehler verwendet werden, ebensowenig, wie ein Logik-Analysator zum Aufsuchen von Fehlern in der Programmlogik dienlich wäre.**

WICHTIGE EIGENSCHAFTEN VON LOGIK-ANALYSATOREN

Logik-Analysatoren variieren in zahlreichen Punkten. Einige hiervon sind:

1) Anzahl der Eingangsleitungen.
Zur Überwachung eines 8-Bit-Datenbusses und eines 16-Bit-Adressenbusses sind wenigstens 24 hiervon erforderlich. Weitere werden für die Steuersignale, Takt und andere wichtige Eingangs-Signale benötigt.

2) Größe des Speichers. Jeder vorausgehende Zustand, der aufbewahrt werden soll, wird mehrere Bytes belegen.
3) Maximale Frequenz. Sie muß mehrere MHz betragen, um auch die schnellsten Prozessoren untersuchen zu können.
4) Minimale Signalbreite (wichtig für das Aufspüren von Spannungsspitzen).
5) Art und Anzahl der zulässigen Trigger-Vorgänge. Wichtige Eigenschaften sind Vor- und Nach-Triggerverzögerungen. Diese gestatten dem Anwender die Anzeige von Vorgängen, die vor oder nach dem Trigger-Ereignis auftreten.
6) Verfahren zur Verbindung mit dem Mikrocomputer. Diese können ziemlich komplexe Interfaces erfordern.
7) Anzahl der Anzeigekanäle.
8) Binäre, hexadezimale oder mnemonische Anzeigen.
9) Anzeigeformate.
10) Anforderung an die Dauer der Aufbewahrung des Signals.
11) Kapazität der Tastköpfe.
12) Einfache oder doppelte Ansprech-Schwellen.

Alle diese Faktoren sind wichtig bein Vergleich verschiedener Logik- und Mikroprozessor-Analysatoren, da diese Instrumente noch neu und nicht standardisiert sind. Eine verwirrende Vielfalt von Produkten ist bereits erhältlich, und diese Vielfalt wird in Zukunft noch zunehmen.

Logik-Analysatoren sind natürlich nur für Systeme mit komplexer zeitlicher Steuerung erforderlich. Einfache Anwendungen mit langsamen peripheren Geräten besitzen wenig Hardware-Probleme, die der Entwickler nicht mit einem Standard-Ozillografen lösen könnte.

FEHLERSUCHE MIT PRÜFLISTEN

Der Entwickler kann möglicherweise nicht ein gesamtes Programm manuell überprüfen. Es gibt jedoch bestimmte Schwerpunkte, die leicht untersucht werden können. **Sie können ein systematisches manuelles Suchen verwenden, um eine große Anzahl von Fehlern aufzufinden, ohne hierzu irgendwelche Fehlersuch-Hilfsmittel zu benötigen.**

WAS ENTHÄLT EINE CHECKLISTE

Die Frage ist, wo die größten Anstrengungen zu machen sind. Die Antwort lautet, an Punkten, die entweder mit einer Ja-Nein-Aussage oder mit einer einfachen arithmetischen Berechnung behandelt werden können. Versuchen Sie nicht, komplexe arithmetische Berechnungen auszuführen, die Flags zu verfolgen oder jeden möglichen Fall zu untersuchen. Beschränken Sie Ihre Überprüfungen auf Gebiete, die leicht zu überblicken sind. Lösen Sie die komplexen Probleme mit Hilfe verschiedener Fehlersuch-Hilfsmittel. Gehen Sie jedoch systematisch vor. Stellen Sie sich Ihre Checkliste auf und vergewissern Sie sich, daß das Programm die grundlegenden Operationen korrekt ausführt.

Der erste Schritt besteht im Vergleichen des Flußdiagramms oder des strukturierten Programmes mit dem tatsächlichen Code. Vergewissern Sie sich, daß alles was in einem hiervon aufscheint, auch im anderen zu finden ist. Eine einfache Checkliste wird diese Aufgabe erfüllen. Es kann sehr leicht vorkommen, daß man eine Verzweigung oder einen Verarbeitungsabschnitt vollkommen übersieht.

Als nächstes konzentrieren Sie sich auf die Programmschleifen. Vergewissern Sie sich, daß alle Register und Speicherplätze innerhalb der Schleifen vor deren Verwendung initialisiert wurden. Dies ist eine häufige Fehlerquelle. Wiederum wird eine einfache Checkliste ausreichen.

Nun sehen Sie sich jede bedingte Verzweigung an. Wählen Sie ein Beispiel, das eine Verzweigung bewirken sollte und eines, bei dem dies nicht der Fall ist. Versuchen Sie beide hiervon. Erfolgt die Verzweigung korrekt oder umgekehrt? Wenn die Verzweigung eine Prüfung beinhaltet, ob sich eine Zahl oberhalb oder unterhalb eines Schwellwertes befindet, versuchen Sie den Fall mit der Gleichheit. Tritt die korrekte Verzweigung auf? Vergewissern Sie sich, daß Ihre Wahl mit der Aufgaben-Definition übereinstimmt.

Sehen Sie sich die Schleifen als Ganzes an. Versuchen Sie die erste und letzte Wiederholung manuell. Diese sind häufig sehr mühsame spezielle Fälle. Was geschieht, wenn die Anzahl der Wiederholungen null ist, d.h., es liegen keine Daten vor, oder die Tabelle besitzt keine Elemente? Läuft das Programm hier korrekt durch? Programme werden häufig eine Wiederholung unnötig ausführen, oder noch ungünstiger, Zähler nach Null dekrementieren, bevor sie diese geprüft haben.

Prüfen Sie alles bis zur letzten Anweisung. Nehmen Sie (hoffnungsvoll) nicht an, daß der erste Fehler auch der einzige in diesem Programm ist. Die manuelle Überprüfung gestattet Ihnen den maximalen Nutzen aus den Fehlersuch-Abläufen zu gewinnen, da Sie nebenbei gleich zahlreiche einfache Fehler loswerden.

FRAGEN BEI MANUELLER PRÜFUNG

Ein kurzer Überblick über die Fragen der manuellen Überprüfung wäre:

1) Befindet sich jedes Element der Programm-Entwicklung im Programm (und umgekehrt für Dokumentationszwecke)?
2) Sind alle Register und Speicherplätze, die innerhalb von Schleifen verwendet werden, vor ihrer Verwendung initialisiert worden?
3) Sind alle bedingten Verzweigungen richtig?
4) Beginnen und enden alle Schleifen ordnungsgemäß?
5) Werden Gleichheits-Fälle korrekt verarbeitet?
6) Werden alle trivialen Fälle ordnungsgemäß verarbeitet?

SUCHEN VON FEHLERN

HÄUFIGE FEHLER

Natürlich arbeiten häufig, ungeachtet dieser Vorsichtsmaßnahmen (es sei denn, Sie haben einige hiervon übergangen), Programme trotzdem nicht. Der Entwickler steht vor der Aufgabe, diese Fehler zu finden. Man kann mit der Prüfliste beginnen, wenn diese nicht bereits verwendet wurde. Einige der Fehler, die Sie vielleicht noch nicht eliminiert haben sind:

1) **Fehler durch mangelnde Initialisierung von Variablen wie Zähler, Zeiger, Summen, Indizes etc.** Nehmen Sie nicht an, daß Register, Speicherplätze oder Flags notwendigerweise Null beinhalten, bevor sie verwendet werden.
2) **Umkehrung eines bedingten Sprunges,** wie die Verwendung von "Verzweige wenn Übertrag gesetzt", wenn Sie meinen "Verzweige wenn Übertrag gelöscht". Achten Sie besonders auf die Tatsache, daß der 6502 (anders als bei den meisten Mikroprozessoren) den Übertrag als ein invertiertes "Borgen" nach einer Subtraktion oder nach einem Vergleich verwendet. So ist der Einfluß eines Vergleiches oder einer Subtraktion wie folgt: (A ist der Inhalt des Akkumulators, M der Inhalt des Speicherplatzes):

Null-Flag = 1 wenn $A = M$
Null-Flag = 0 wenn $A \neq M$
Übertrags-Flag = 1 wenn $A \geq M$
Übertrags-Flag = 0 wenn $A < M$

Beachten Sie besonders, daß Übertrag = 1 wenn A = M (der Gleichheits-Fall). Hier bedeutet "Verzweige wenn Übertrag gesetzt" einen Sprung, wenn $A \geq M$ und "Verzweige wenn Übertrag gelöscht" bedeutet einen Sprung, wenn $A < M$. Wenn Sie den Gleichheits-Fall auf der anderen Seite wünschen, versuchen Sie, die Rollen von A und M zu vertauschen, oder addieren Sie 1 zu M. Wenn Sie beispielsweise einen Sprung für den Fall $A \geq 10$ wollen, so verwenden Sie

```
CMP   #10
BCC   ADDR
```

Wenn Sie andereseits einen Sprung für den Fall $A > 10$ wollen, verwenden Sie:

```
CMP   #10
BCS   ADDR
```

3) **Weiterstellen der Zähler und Zeiger an der falschen Stelle oder überhaupt nicht.** Vergewissern Sie sich, ob es keine Wege durch eine Schleife gibt, bei denen entweder die Befehle für das Weiterstellen übersprungen oder wiederholt werden.
4) **Ausfallen von korrekten Durchläufen in trivialen Fällen,** wie keine Daten in einem Puffer, keine Tests auszuführen, oder keine Eingaben bei der Übertragung. Nehmen Sie niemals an, daß der derartige Fälle nie auftreten, außer das Programm hat diese Fälle speziell eliminiert.

Andere Probleme auf die zu achten sind:

5) **Umkehrung der Reihenfolge von Operanden.** Erinnern Sie sich daran, daß zum Beispiel der TAX-Befehl A zu X bringt, jedoch nicht umgekehrt.
6) **Änderung von Bedingungsflags, bevor sie verwendet werden.** Erinnern Sie sich daran, daß alle Befehle, außer Speicher- und Verzweigungsbefehlen die Flags beeinflussen, insbesondere Vorzeichen- und Null-Flags. Erinnern Sie sich auch daran, daß PLP und RTI alle Flags ändern können.
7) **Verwechslung des Indexregisters und des indizierten Speicherplatzes.** Beachten Sie, daß INX und INY das Indexregister inkrementieren, während INC ADDR,X und andere ähnliche Befehle den Inhalt der indizierten Speicherplatzes inkrementieren.
8) **Verwechslung von Daten und Adressen.** Erinnern Sie sich daran, daß LDA #$40 in A die Zahl 40_{16} lädt, während LDA $40 den Akkumulator mit dem Inhalt des Speicherplatzes 0040_{16} lädt. Seien Sie vorsichtig, wenn Sie die vorindizierte und nachindizierte Adressierung verwenden, bei denen ein Paar von Adressen auf der Nullseite die tatsächliche oder Basis-Adresse der Daten enthält.
9) **Zufällige Re-Initialisierung eines Registers oder Speicherplatzes.** Vergewissern Sie sich, daß kein Sprungbefehl die Steuerung zurück zu Initialisierungs-Anweisungen transferiert.
10) **Verwechslung von Zahlen und Zeichen.** Erinnern Sie sich daran, daß die ASCII und EBCDIC-Darstellungen von Ziffern sich von den Ziffern selbst unterscheicen. Beispielsweise ist ASCII 7 gleich 37_{16}, wogegen 07_{16} das Klingelzeichen (BELL) in ASCII ist.
11) **Verwechslung von Binär- und Dezimalzahlen.** Erinnern Sie sich daran, daß die BCD-Darstellung einer Zahl verschieden von seiner Binär-Darstellung ist. Beispielsweise ist BCD 36 gleich binär 54 (versuchen Sie es).
12) **Umkehrung der Reihenfolge bei einer Subtraktion. Seien Sie auch bei anderen Operationen sehr vorsichtig (wie der Division), daß Sie diese nicht vertauschen.** Erinnern Sie sich daran, daß SBC, CMP, CPX und CPY alle den Inhalt des adressierten Speicherplatzes **vom** Inhalt des Akkumulators oder Indexregisters subtrahieren.
13) **Ignorieren der Effekte von Unterprogrammen und Makros.** Nehmen Sie nicht an, daß das Aufrufen von Unterprogrammen oder Bezugnahme auf Makros die Flags, Register oder Speicherplätze nicht ändert. Vergewissern Sie sich genau, welche Auswirkungen sie haben. Beachten Sie, daß es sehr wichtig ist, diese Einflüsse schriftlich festzuhalten.
14) **Falsche Verwendung von Schiebebefehlen.** Erinnern Sie sich an die genauen Auswirkungen von ASL, LSR, ROL und ROR. Diese sind 1-Bit-Verschiebungen, die das Übertrags-, Vorzeichen- und Null-Flag beeinflussen. ASL und LSR löschen beide das leere Bit. ROR und ROL sind zirkulare Verschiebungen, die den Übertrag mit einschließen. Erinnern Sie sich daran, daß das Übertrags-, Vorzeichen- und Null-Flag beeinflußt wird, auch wenn diese Befehle auf Daten in einem Speicherplatz angewendet werden.

15) **Falsches Zählen der Länge einer Anordnung.**
Erinnern Sie sich daran, daß es fünf (nicht vier) Speicherplätze gibt, die in den Adressen 0300 bis 0304 (einschließlich) enthalten sind.

16) **Verwechslung von 8- und 16-Bit-Größen.**
Adressen sind tatsächlich 16 Bits lang. Das einzige Register des 6502, das eine vollständige Adresse aufbewahren kann, ist der Befehlszähler.

17) **Vergessen, daß 16-Bit-Zahlen zwei Speicherplätze belegen.**
Absolute direkte oder absolute indizierte Adressierung belegt zwei Speicherplätze, so wie es die Adressen machen, die auf der Nullseite gespeichert sind, zur Verwendung in der nachindizierten oder vorindizierten Adressierung. Der Befehlszähler belegt auch zwei Speicherplätze, wenn er im Stapel gespeichert wird. Beachten Sie, daß bei der vorindizierten und nachindizierten Adressierung zwei Speicherplätze verwendet werden, obwohl nur einer spezifiziert wird. Die Adresse, die unmittelbar der einen spezifierten folgt, wird auch zum Aufbewahren der indirekten Adresse benötigt.

18) **Verwechseln des Stapels und des Stapelzeigers.**
Der Befehl TXS beeinflußt den Stapelzeiger, nicht den Inhalt des Stapels. PHA, PHP und PLP transferieren Daten zum oder vom Stapel. Erinnern Sie sich daran, daß JSR, RTS, RTI und BRK ebenfalls den Stapel verwenden. Erinnern Sie sich ferner daran, daß Sie den Stapelzeiger initialisieren müssen, bevor Sie irgend ein Unterprogramm aufrufen oder Unterbrechungen gestatten. Der Stapel des 6502 liegt immer auf Seite 1; nur die 8 letzten signifikanten Bits der Stapeladresse liegen tatsächlich im Stapelzeiger.

19) **Änderung eines Registers oder Speicherplatzes vor dessen Verwendung.**
Erinnern Sie sich daran, daß LDA, STA, LDX, STX, LDY, STY, TAX, TXA etc. den Inhalt des Bestimmungsortes (jedoch nicht der Quelle) ändern.

20) **Vergessen des Transferierens der Steuerung nach Abschnitten des Programmes, die in bestimmten Situationen nicht ausgeführt werden sollten.**
Erinnern Sie sich daran, daß der Computer sequentiell durch den Programmspeicher gehen wird, außer es wird ihm speziell anders vorgeschrieben.

21) **Vergessen, daß der Übertrag immer in Additions- und Subtraktions-Operationen enthalten ist.**
Der 6502 besitzt nur Befehle "Addiere mit Übertrag" und "Subtrahiere mit Borgen", anders als bei zahlreichen anderen Prozessoren, die reguläre Additions- und Subtraktionsbefehle besitzen, die keinen Übertrag beinhalten. Der Übertrag muß ausdrücklich vor einer Addition gelöscht und vor einer Subtraktion gesetzt werden, wenn sein Wert nicht von der Operation beeinflußt wird. Beachten Sie jedoch, daß die Vergleichsbefehle (CMP, CPX, CPY) keinen Übertrag beinhalten.

22) **Invertieren des Vorzeichens des Übertrags bei einer Subtraktion.**
Bei Subtraktions- und Vergleichsbefehlen ist der sich ergebende Übetrag ein invertiertes "Borgen", d.h., der Übertrag wird gesetzt, wenn kein Borgen erforderlich ist. Entsprechend subtrahiert der Befehl "Subtrahiere mit Borgen" den invertierten Übertrag (1 – Übetrag), zusammen mit dem Inhalt des spezifizierten Speicherplatzes.

23) **Falsche Verwendung der dezimalen Betriebsart.**
Wenn das Flag "Decimal Mode" gesetzt ist, sind alle arithmetischen Ergebnisse dezimal. Daher muß das Flag ausdrücklich gelöscht werden, wenn die Dezimal-Operationen abgeschlossen sind. Andernfalls wird es die Ergebnisse von Operationen ändern, die nicht in dezimal ausgeführt werden sollen. Beachten Sie, daß alle Wege, die einen Befehl "Setze Dezimal-Betriebsart" beinhalten, auch einen Befehl "Lösche Dezimal-Betriebsart" haben müssen. Seien Sie besonders sorgfältig bei einem "Fall-through" und bei Austritten infolge eines Fehlers.

24) **Falsche Verwendung des Befehls "Bit-Test".**
Beachten Sie, daß der Bit-Test-Befehl das Vorzeichen- und Überlauf-Flag entsprechend der Bits 7 und 6 des geprüften Speicherplatzes setzt, ohne Rücksicht auf den Inhalt des Akkumulators. Dieser Befehl ist sehr bequem zum Testen der Statusbits im PIA 6502 und anderen Bitprüf-Operationen, erfordert jedoch sorgfältige Dokumentation, da seine Ergebnisse oft unklar für einen Leser sein können.

FEHLERSUCHE IN UNTERBRECHUNGS-GESTEUERTEN PROGRAMMEN

Unterbrechungs-gesteuerte Programme sind besonders schwierig fehlerfrei zu machen, da Fehler zufällig auftreten können. Wenn zum Beispiel das Programm die Unterbrechungen einige Befehle zu früh freigibt, wird ein Fehler nur dann auftreten, wenn eine Unterbrechung gerade bei Ausführung dieser wenigen Befehle empfangen wird.
In der Tat können Sie für gewöhnlich annehmen, daß zufällig auftretende Fehler durch das Unterbrechungssystem bewirkt werden. Typische Fehler in unterbrechungs-gesteuerten Programmen sind:

1) **Vergessen der Wieder-Freigabe von Unterbrechungen nach der Annahme und Bedienung einer Unterbrechung.**
Der Prozessor sperrt das Unterbrechungssystem automatisch bei RESET oder bei der Annahme einer Unterbrechung. Vergewissern Sie sich, daß keine möglichen Sequenzen versäumen, die Unterbrechung wieder freizugeben.
2) **Verwendung des Akkumulators bevor er aufbewahrt wird.** D.h. PHA muß jeder Operation vorausgehen, die den Akkumulator ändert.
3) **Vergessen des Aufbewahrens und Zurückspeicherns des Akkumulators.**
4) **Zurückspeichern der Register in falscher Reihenfolge.**
Wenn die Reihenfolge, in der die Register aufbewahrt wurden, war:

```
PHA     ;BEWAHRE AKKUMULATOR-INHALT AUF
TXA     ;BEWAHRE INDEXREGISTER X AUF
PHA
TYA     ;BEWAHRE INDEXREGISTER Y AUF
PHA
```

so sollte die Reihenfolge des Zurückspeicherns sein:

```
PLA     ;SPEICHERE INDEXREGISTER Y ZURÜCK
TAY
PLA     ;SPEICHERE INDEXREGISTER X ZURÜCK
TAX
PLA     ;SPEICHERE AKKUMULATOR-INHALT ZURÜCK
```

5) **Freigabe von Unterbrechungen, bevor alle erforderlichen Bedingungen eingerichtet wurden,** wie Priorität, Flags, PIA-und VIA-Konfigurationen, Zeiger, Zähler, etc. Eine Checkliste kann hier helfen.

6) **Belassen von Ergebnissen in Register und deren Zerstörung beim Rückspeicher-Vorgang.**
Wie früher erwähnt sollten die Register nicht zum Transferieren von Informationen zwischen dem Programm und den Unterbrechungs-Serviceroutinen verwendet werden.

7) **Vergessen, daß die Unterbrechungen (einschließlich BRK) den alten Befehlszähler und das Statusregister im Speicher lassen, ob Sie diesen verwenden oder nicht.**
Sie müssen den Stapelzeiger reininitialisieren oder auf den neuesten Stand bringen.
8) **Ignorieren der Möglichkeit, daß in die Serviceroutine mit gesetztem Flag für die Dezimal-Betriebsart eingetreten wird.**
Sie müssen einen CLD-Befehl in die Service-Routine einschließen, wenn diese Möglichkeit vorkommt. Beachten Sie, daß RTI automatisch den ursprünglichen Zustand des Flags am Ende der Serviceroutine wiederherstellen wird.
9) **Nicht-Freigeben der Unterbrechung während Mehr-Wort-Transfers oder Befehlssequenzen.**
Achten Sie sorgfältig auf Situationen, bei denen Unterbrechungs-Serviceroutinen die gleichen Speicherplätze wie das Programm verwenden könnte.

Hoffentlich gibt Ihnen diese Liste annähernd eine Vorstellung worauf Sie achten sollten. Unglücklicherweise kann ein noch so systematisches Fehlersuchen noch genügend ungelöste Probleme zurücklassen, speziell wenn Unterbrechungen im Programm vorkommen[3].

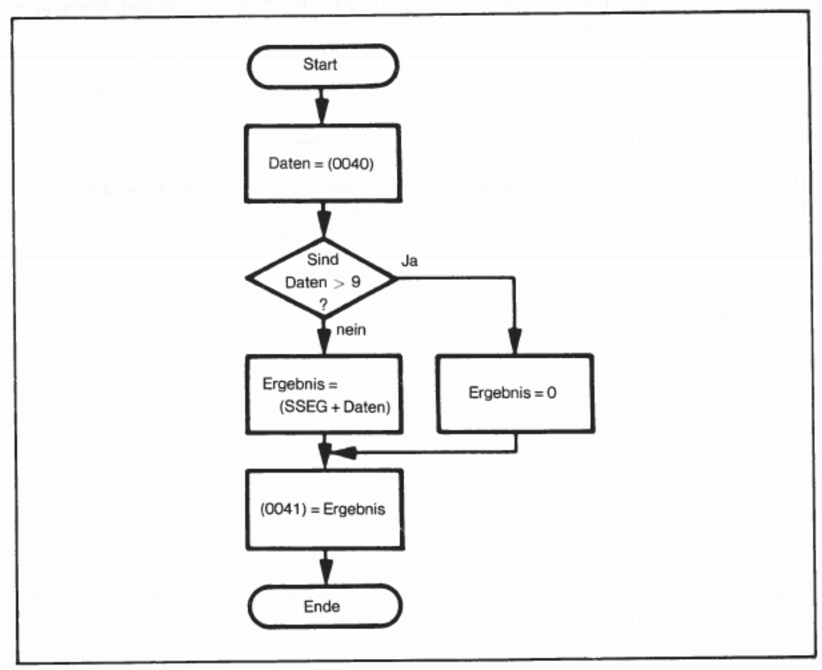


Bild 14-5. Flußdiagramm der dezimal in Sieben-Segment-Umwandlung

Fehlersuch-Beispiel 1: Dezimal/Sieben-Segment-Umwandlung

FEHLERSUCHE IN EINEM CODE-UMWANDLUNGS-PROGRAMM

Das Programm wandelt eine Dezimalzahl im Speicherplatz 0040 in einen Sieben-Segment-code im Speicherplatz 0041 um. Es steuert die Anzeige dunkel, wenn der Speicherplatz 0040 keine Dezimalzahl enthält.

Anfängliches Programm (vom Flußdiagramm in Bild 14-5):

```
        LDX    #$40      ;HOLE DATEN
        CPX    #9        ;SIND DATEN GRÖSSER ALS 9?
        BCC    DONE      ;JA, ERLEDIGT
        LDAA   (SSEG,X)  ;HOLE ELEMENTE VON TABELLE
DONE    BRK
SSEG    .BYTE            $3F,$06,$5B,$4F,$66
        .BYTE  .BYTE     $6D,$7D,$07,$7D,$6F
```

Unter Verwendung der Checkliste sind wir imstande, folgende Fehler aufzufinden:

1) Der Block, der das Ergebnis löschte, wurde entfernt.
2) Die bedingte Verzweigung war falsch.

Wenn zum Beispiel die Daten null sind, so löscht CPX #9 den Übertrag, da 0 < 9 und ein "Borgen" erforderlich ist. Der Sprung mit der entgegegesetzten Bedingung jedoch, (d.h., BCS DONE) erzeugt noch nicht das richtige Ergebnis. Nun verarbeitet das Programm den Gleichheitsfall inkorrekt da, wenn die Daten 9 sind, der Befehl CMX #9 den Übertrag löscht und einen Sprung verursacht. Die richtige Version ist:

```
        CPX    #10       ;SIND DATEN EINE DEZIMALZAHL?
        BCS    DONE      ;NEIN, BEWAHRE FEHLERCODE AUF
```

Zweites Programm:

```
        LDA    #0        ;HOLE CODE FÜR BLANK-(LEER) ZEICHEN
                         ;  FÜR DIE ANZEIGE
        LDX    #$40      ;HOLE DATEN
        CPX    #10       ;SIND DATEN EINE DEZIMALZAHL?
        BCS    DONE      ;NEIN, BEWAHRE FEHLERCODE AUF
        LDA    (SSEG,X)  ;HOLE ELEMENT VON TABELLE
        STA    $41       ;BEWAHRE SIEBEN-SEGMENT- ODER
                         ;  FEHLERCODE AUF
DONE    BRK
SSEG    .BYTE            $3F,$06,$5B,$4F,$66
        .BYTE            $6D,$7D,$07,$7D,$6F
```

Diese Version wurde manuell mit Erfolg geprüft.

Da das Programm einfach war, wurde es als nächstes mit Einzelschritten mit realen Daten durchlaufen. Die für den Versuch gewählten Daten waren:

0	(die kleinste Zahl)
9	(die größte Zahl)
10	(ein Grenzfall)
$6B_{16}$	(willkürlich)

Der erste Versuch geschah mit Null im Speicherplatz 0040. Der erste Fehler war offensichtlich – LDX#$40, lud die Zahl 40 in X, nicht den Inhalt des Speicherplatzes 0040. Der richtige Befehl war LDX $40 (direkte anstatt unmittelbare Adressierung). Nachdem diese Korrektur ausgeführt wurde, arbeitete das Programm weiter mit keinen augenscheinlichen Fehlern, bis es den Befehl LDA (SSEG,X) auszuführen versuchte.

Der Inhalt des Adressenbusses während des Holens der Daten war 063F, eine Adresse, die bisher nicht verwendet wurde. Offensichtlich ging irgend etwas falsch.

Es ist nun Zeit für weitgehende manuelle Prüfung. Da wir wußten, daß BCS DONE korrekt war, lag der Fehler offensichtlich im LDA-Befehl. Eine manuelle Prüfung zeigte:

LDA (SSEG,X) addiert den Inhalt des Indexregisters X zur Nullseiten-Adresse SSEG und verwendet die Summe, um die Adresse zu holen, die die tatsächlichen Daten enthalten. Im vorliegenden Fall, da das Register X Null enthält, liegt die indirekte Adresse im Speicherplatz SSEG und SSEG+1, d.h., sie ist 063F. Der Befehl holt daher eine Adresse aus einer Tabelle, die aus Daten besteht. Der richtige Befehl ist LDA SSEG,X – wir wollen Daten von der Tabelle holen, nicht die Adresse der Daten.

Auch mit dieser Berichtigung erzeugte das Programm ein Ergebnis mit Null, anstatt des erwarteten 3F. Der Fehler lag offensichtlich im letzten Befehl – er sollte STA $41 und nicht STX $41 sein. Beachten Sie, wie wichtig es ist, das Programm wirklich bis zum Ende zu verfolgen, anstatt nach dem scheinbar letzten Fehler aufzuhören.

Das revidierte Programm war nun:

Drittes Programm:

```
        LDA    #0         ;HOLE FEHLER CODE FÜR DIE ANZEIGE
        LDX    $40        ;HOLE DATEN
        CPX    #10        ;SIND DIE DATEN EINE DEZIMALZAHL?
        BCS    DONE       ;NEIN, BEWAHRE FEHLERCODE AUF
        LDA    SSEG,X     ;HOLE ELEMENT VON TABELLE
        STA    $41        ;BEWAHRE SIEBEN-SEGMENT-CODE ODER
                          ;  FEHLERCODE AUF
DONE    BRK
SSEG    .BYTE             $3F,$06,$5B,$4F,$66
        .BYTE             $6D,$7D,$07,$7D,$6F
```

Dieses Programm erzeugte die folgenden Ergebnisse:

Daten	Ergebnis
00	3F
09	6F
0A	6F
6B	6F

Dieses Programm löschte nicht das Ergebnis, wenn die Daten ungültig waren, d.h., größer als 9. Das Programm speichert niemals den "Blank-Code", da die Bestimmungsadresse DONE an der falschen Stelle lag – sie sollten dem Befehl STA $41 zugeordnet werden. Nachdem diese Korrekturen ausgeführt wurden, erbrachte das Programm korrekte Resultate für alle Testfälle.

Da das Programm einfach war, könnte es für alle Dezimalziffern geprüft werden. Die Ergebnisse wären:

Daten	Ergebnis
0	3F
1	06
2	5B
3	4F
4	66
5	6D
6	7D
7	07
8	7D
9	6F

Beachten Sie, daß das Ergebnis für die zahl 8 falsch ist, es sollte 7F sein. Da alles andere korrekt ist, liegt der Fehler ziemlich sicher in der Tabelle. In der Tat, ist die Eintragung 8 in der Tabelle falsch.

Das endgültige Programm ist:

```
;
;DEZIMAL IN SIEBEN-SEGMENT-UMWANDLUNG
;
        LDA    #0         ;HOLE "BLANK"-CODE FÜR DIE ANZEIGE
        LDA    $40        ;HOLE DATEN
        CPX    #10        ;SIND DIE DATEN EINE DEZIMALZAHL?
        BCS    DONE       ;NEIN, BEWAHRE FEHLER-CODE AUF
        LDA    SSEG,X     ;HOLE SIEBEN-SEGMENT-CODE VON
                          ;  TABELLE
DONE    STA    $41        ;BEWAHRE SIEBEN-SEGMENT-CODE ODER
                          ;  FEHLER-CODE AUF
SSEG    .BYTE             $3F,$06,$5B,$4F,$66
        .BYTE             $6D,$7D,$07,$7F,$6F
```

Derartige in diesem Programm aufgetretene typische Fehler sollten von Programmierern, die die Assemblersprache des 6502 verwenden, vermieden werden. Sie beinhalten:

1) Vergessen der Initialisierung von Registern oder Speicherplätzen.
2) Invertieren der Logik von bedingten Verzweigungen.
3) Falsche Verzweigung im Gleichheitsfall der Operanden.
4) Verwechslung unmittelbarer und direkter Adressierung, d.h. Daten und Adressen.
5) Vergessen, den momentanen Inhalt von Registern zu verfolgen.
6) Verzweigung zur falschen Stelle, so daß ein bestimmter Weg durch das Programm falsch ist.
7) Falsches Kopieren von Listen und Zahlen.
8) Falsche Verwendung der indirekten Adressierung.

Beachten Sie daß geradlinige Befehle (wie AND, DEC, INC) und einfache Adressier-Arten selten irgendwelche Probleme zur Folge haben. Unter speziell unangenehmen Fehlern, die häufig in der Assembler-Programmiersprache des 6502 vorkommen, ist die falsche Verwendung des Übertrags nach Subtraktionen oder Vergleichen (der Übertrag wird gesetzt, wenn kein "Borgen" erforderlich ist) und Vergessen das Flag für die Dezimal-Betriebsart zu löschen.

Fehlersuch-Beispiel 2: Sortieren in absteigender Reihenfolge

FEHLERSUCHE IN EINEM SORTIER-PROGRAMM

Das Programm sortiert eine Anordnung von Binärzahlen mit 8 Bits und ohne Vorzeichen in absteigender Reihenfolge. Die Anordnung beginnt im Speicherplatz 0041 und seine Länge liegt im Speicherplatz 0040.

Anfangsprogramm (vom Flußdiagramm in Bild 14-6):

```
        LDY   #0      ;LÖSCHE AUSTAUSCH FLAG VOR DEM
                      ;  DURCHLAUF
        LDX   $40     ;HOLE LÄNGE DER ANORDNUNG
PASS    LDA   $41,X   ;BEFINDET SICH DAS NÄCHSTE PAAR IN
                      ;  RICHTIGER REIHENFOLGE?
        CMP   $42,X
        BCC   COUNT   ;JA, KEIN AUSTAUSCH ERFORDERLICH
        STA   $42,X   ;NEIN, TAUSCHE PAAR AUS
COUNT   DEX           ;PRÜFE AUF VOLLSTÄNDIGEN DURCHLAUF
        BNE   PASS
        DEY           ;SIND ALLE ELEMENTE IN DER RICHTIGEN
                      ;  REIHENFOLGE?
        BNE   PASS    ;NEIN, MACHE EINEN WEITEREN DURCH-
                      ;  LAUF
        BRK
```

Die manuelle Überprüfung zeigt, daß alle Blöcke in dem Flußdiagramm im Programm ausgeführt wurden und daß alle Register initialisiert wurden. Die bedingten Verzweigungen müssen sorgfältig geprüft werden. Der Befehl BCC COUNT muß eine Verzweigung erzwingen, wenn der neue Wert in A größer oder gleich dem nächsten Wert in der Anordnung ist. Erinnern Sie sich daran, daß wir Elemente in absteigender Reihenfolge sortieren und daß wir uns durch die Anordnung auf die gebräuchliche Art und Weise des 6502 rückwärts bewegen. Beachten Sie, daß der Gleichheitsfall nicht in einem Austausch resultieren darf, da dies eine endlose Schleife erzeugen würde, in der die beiden gleichen Elemente immer wieder ausgetauscht werden.

Versuchen Sie ein Beispiel:

(0041) = 30
(0042) = 37

CMP $42,X resultiert in der Berechnung von 30 - 37. Der Übertrag wird gelöscht, da ein "Borgen" erforderlich ist. Dieses Beispiel sollte in einem Austausch resultieren, tut es jedoch nicht.

BCS COUNT wird die richtige Verzweigung in diesem Fall liefern. Wenn die beiden Zahlen gleich sind, wird der Vergleich den Übertrag setzen und BCS COUNT ist wieder richtig.

Wie sieht es mit BNE PASS am Ende des Programmes aus? Wenn es irgend ein Element außerhalb der Reihenfolge gibt, wird das Austauschflag Eins sein, so daß die Verzweigung falsch ist. Sie sollte BEQ PASS sein.

Nun versuchen Sie den ersten Durchlauf. Die Initialisierung sollte in folgendem resultieren:

```
          X = LÄNGE (2)
          Y = 0
```

Die Auswirkungen der Schleifen-Befehle sind:

```
        LDA   $41,X        ;A = (0043)
        CMP   $42,X        ;(0043)-(0041)
        BCS   COUNT
        STA   $42,X        ;(0044) = (0043)
COUNT   DEX                ;X = LÄNGE - 1 (1)
        BNE   PASS
```

Die indizierten Adressen sind offensichtlich falsch, da sie beide hinter dem Ende der Anordnung liegen. Wir wollen sie ändern, indem wir zwei von den Adressen subtrahieren, die in den indizierten Befehlen enthalten sind. Diese Versetzung ist ein häufiges Problem in Assemblersprachen-Programmen des 6502, da Anordnungen mit fünf Elementen die Speicheradressen BASE bis BASE+4 belegen und nicht BASE+1 bis BASE+5. Wenn die indizierte Adressierung beim Mikroprozessor 6502 verwendet wird, so achten Sie sehr sorgfältig darauf, daß Ihre Adressen an einem oder anderen Ende der Anordnung nicht fehlerhaft sind.

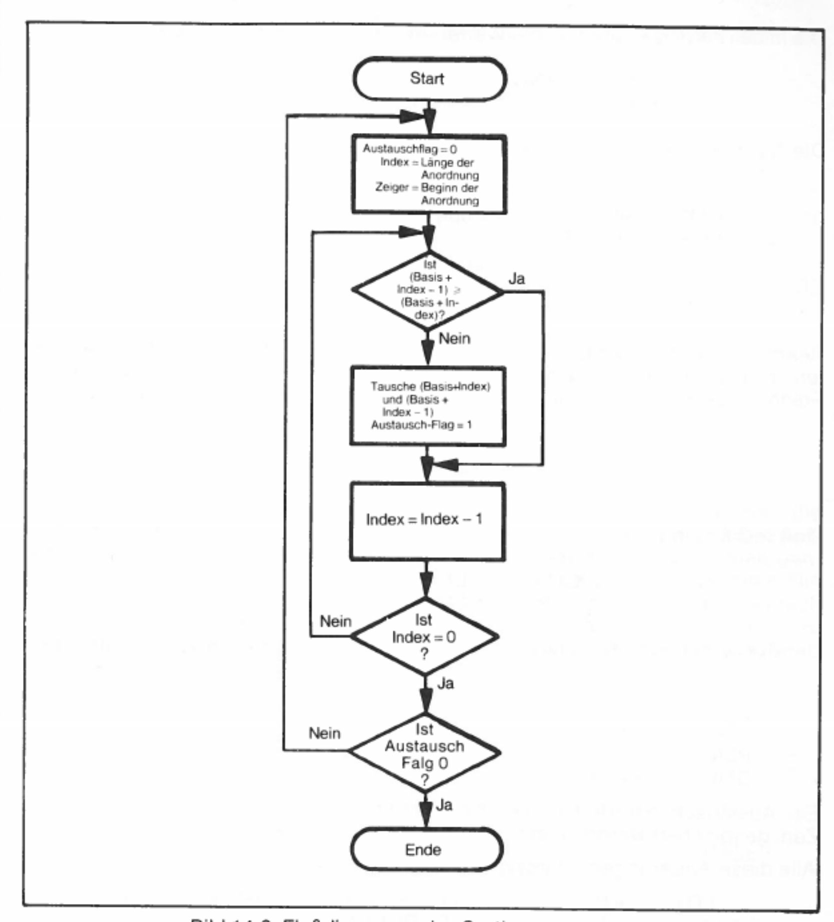


Bild 14-6. Flußdiagramm des Sortierprogramms

Die Initialisierung ergibt nun die Werte:

```
X = LÄNGE (2)
Y = 0
```

Die Auswirkungen der Schleifenbefehle sind:

```
        LDA   $3F,X      ;A = (0041)
        CMP   $40,X      ;(0041)-(0042)
        BCS   COUNT
        STA   $40,X      ;(0042) = (0041)
COUNT   DEX              ;X = LÄNGE - 1 (1)
        BNE   PASS
```

Beachten Sie, daß wir bereits die bedingten Sprungbefehle geprüft haben. Offensichtlich ist die Logik falsch. Wenn die ersten beiden Elemente außerhalb der Reihenfolge liegen, so sollten die Resultate nach der ersten Wiederholung sein:

```
(0041) = ALT (0042)
(0042) = ALT (0041)
     X = LÄNGE - 1
```

Stattdessen sind sie:

```
(0041) = UNVERÄNDERT
(0042) = ALT (0041)
     X = LÄNGE - 1
```

Der Austausch erfordert etwas mehr Sorgfalt und die Verwendung des Stapels:

```
PHA
LDA   $40,X
STA   $3F,X
PLA
STA   $40,X
```

Ein Austausch erfordert immer einen zeitweiligen Speicherplatz, in den eine Zahl gespeichert werden kann, während die andere transferiert wird.[4]

Alle diese Änderungen erfordern ein neues Programm:

```
        LDY   #0         ;LÖSCHE AUSTAUSCHFLAG VOR DEM
                         ;  DURCHLAUF
        LDX   $40        ;HOLE LÄNGE DER ANORDNUNG
PASS    LDA   $3F,X      ;IST DAS NÄCHSTE PAAR IN RICHTIGER
                         ;  REIHENFOLGE?
        CMP   $40,X
        BCS   COUNT      ;JA, KEIN AUSTAUSCH ERFORDERLICH
        PHA              ;NEIN, TAUSCHE ELEMENTE UNTER VER-
                         ;  WENDUNG DES STAPELS AUS
        LDA   $40,X
        STA   $3F,X
        PLA
        STA   $40,X
COUNT   DEX              ;IST DURCHLAUF VOLLSTÄNDIG?
        BNE   PASS
        DEY              ;SIND ALLE ELEMENTE IN DER RICHTIGEN
                         ;  REIHENFOLGE?
        BEQ   PASS       ;NEIN, MACHE EINEN WEITEREN DURCH-
                         ;  LAUF
        BRK
```


Wie sieht es mit der letzten Wiederholung aus? Nehmen wir an, daß hier drei Elemente vorliegen:

```
(0040) = 03   (Anzahl der Elemente)
(0041) = 02
(0042) = 04
(0043) = 06
```

Bei jedem Durchlauf inkrementiert das Programm X um 1. Daher ist bei Beginn der dritten Wiederholung (X) = 1. Die Auswirkung der Schleifen-Befehle sind:

```
LDA   $3F,X      ;(A) = (0040)
CMP   $40,X      ;(0040) - (0041)
```

Dies ist nicht korrekt. Das Programm hat versucht, jenseits der Startadresse der Daten weiterzulaufen. Die vorhergehende Wiederholung sollte in der Tat die letzte sein, da die Anzahl der Paare um 1 kleiner ist als die Anzahl der Elemente. Die Korrektur besteht im Reduzieren der Anzahl von Wiederholungen um 1; dies kann durch Plazieren von DEX nach LDX $40 geschehen. Wir müssen daher 1 zu allen Adressen in den indizierten Befehlen addieren.

Wie steht es mit den trivialen Fällen? Was geschieht, wenn die Anordnung überhaupt keine Elemente enthält, oder nur ein Element? Die Antwort lautet, daß das Programm nicht ordnungsgemäß arbeitet und einen ganzen Datenblock ohne irgendeine Warnung verfälschen kann (versuchen Sie es). Die Korrekturen zur Handhabung der trivialen Fälle sind einfach aber wesentlich. Sie kosten nur einige wenige Speicherplätze um Probleme zu vermeiden, die später schwer aufzufinden wären. Das neue Programm lautet:

```
        LDY   #0         ;LÖSCHE AUSTAUSCHFLAG VOR DEM
                         ;  DURCHLAUF
        LDX   $40        ;HOLE LÄNGE DER ANORDNUNG
        CPX   #2         ;HAT DIE ANORDNUNG 2 ODER MEHR ELE-
                         ;  MENTE?
        BCC   DONE       ;NEIN, KEINE AKTION ERFORDERLICH
        DEX              ANZAHL DER PAARE = LÄNGE - 1
PASS    LDA   $40,X      ;IST DAS NÄCHSTE PAAR IN RICHTIGER
                         ;  REIHENFOLGE?
        CMP   $41,X
        BCS   COUNT      ;JA, KEIN AUSTAUSCH ERFORDERLICH
        PHA              ;NEIN, TAUSCHE ELEMENTE UNTER VER-
                         ;  WENDUNG DES STAPELS AUS
        LDA   $41,X
        STA   $40,X
        PLA
        STA   $41,X
COUNT   DEX              ;IST DER DURCHLAUF VOLLSTÄNDIG?
        BNE   PASS
        DEY              ;WAREN ALLE ELEMENTE IN RICHTIGER
                         ;  REIHENFOLGE?
        BEQ   PASS       ;NEIN, MACHE EINEN WEITEREN DURCH-
                         ;  LAUF
DONE    BRK
```

Nun ist es an der Zeit, das Programm auf einem Computer oder auf einem Simulator zu prüfen. Ein einfacher Datensatz wäre:

(0040)	=	02	Länge der Anordnung
(0041)	=	00	zu sortierende Anordnung
(0042)	=	01	

Dieser Satz besteht aus zwei Elementen in der falschen Reihenfolge. Das Programm sollte zwei Durchläufe ausführen. Der erste Durchlauf sollte die Elemente neu anordnen, wobei erzeugt wird:

(0041)	=	01	neu angeordnete Anordnung
(0042)	=	00	
Y	=	01	Austausch-Flag

Der zweite Durchlauf sollte die Operation abschließen und erzeugen:

Y	=	00	Austausch-Flag

Dieses Programm ist ziemlich lang, um es in Einzelschritten zu durchlaufen, so daß wir stattdessen Haltepunkte verwenden wollen. Jeder Haltepunkt wird den Computer anhalten und den Inhalt aller Register ausdrucken. Die Haltepunkte werden kommen:

1) Nach DEX um die Anfangsbedingungen zu prüfen.
2) Nach CMP $41,X um den Vergleich zu prüfen.
3) Nach STA $41,X um den Austausch zu prüfen.
4) Nach DEY, um den Abschuß eines Durchlaufes durch die Anordnung zu prüfen.

Der Inhalt der Register nach dem ersten Durchlauf wäre:

Register	Inhalt
X	01
Y	00
P (Status)	25 (35, wenn Sie BRK für den Haltepunkt verwenden, da das BRK-Flag gesetzt wird.)

Diese sind alle korrekt, so daß das Programm die Anfangsbedingungen in diesem Fall richtig berechnete.

Die Ergebnisse beim zweiten Haltepunkt wären:

Register	Inhalt
A	00
X	01
Y	00
P (Status)	A4 (B4 wenn Sie BRK verwenden)

Diese Ergebnisse sind ebenfalls korrekt.

Die Ergebnisse beim dritten Haltepunkt wären:

Register	Inhalt
A	00
X	01
Y	00
P (Status	26 (36 wenn Sie BRK verwenden)

Ein Prüfen des Speichers zeigte:

(0041)	=	01
(0042)	=	00

Die Ergebnisse beim vierten Haltepunkt wären:

Register	Inhalt
A	00
X	00
Y	FF
P (Status)	A4 (B4 wenn Sie BRK verwenden)

Hier ist das Null-Flag (Bit 1 des Status-Registers) nicht korrekt, das anzeigt, daß kein Austausch aufgetreten ist. Das Register Y enthält nicht den richtigen Wert – es sollte auf 1 nach dem Austausch gesetzt werden. In der Tat zeigt ein Blick auf das Programm, daß kein Befehl jemals das Indexregister Y änderte, um das Auftreten eines Austausches zu markieren. Die Korrektur besteht hier im Plazieren des Befehls LDY #1 nach BCS COUNT.

Nun besteht das Verfahren im Laden des Index-Registers Y mit dem korrekten Wert (Null), Setzen des Null-Flags auf 1 und der Fortsetzung. Die zweite Wiederholung des zweiten Haltepunktes ergibt:

Register	Inhalt
A	02
X	00
Y	00
P (Status)	25 (35 wenn Sie BRK verwenden)

Offensichtlich ist das Programm unkorrekt weitergelaufen, ohne die Register neu zu initialisieren (speziell das Indexregister X). Der bedingte Sprung, der vom Austausch-Flag abhängt, sollte die Steuerung auf jeden Fall zu einem Punkt des Programmes zurück transferieren, der X reininitialisiert.

Beachten Sie, daß wir weder Y reinintialisieren brauchen (es wird immer Null sein – weshalb?), noch die Länge der Anordnung neuerlich prüfen müssen.

Die endgültige Version des Programmes ist:

```
SORT    LDY   #0       ;LÖSCHE AUSTAUSCHFLAG ZU BEGINN
        LDX   $40      ;GIBT ES ZWEI ODER MEHR ELEMENTE?
        CPX   #2
        BCC   DONE     ;NEIN, KEINE AKTION NÖTIG
ITER    LDX   $40      ;JA, ANZAHL DER PAARE = LÄNGE - 1
        DEX
PASS    LDA   $40,X    ;IST DAS NÄCHSTE PAAR IN RICHTIGER
                       ;  REIHENFOLGE?
        CMP   $41,X
        BCS   COUNT    ;JA, KEIN AUSTAUSCH ERFORDERLICH
        LDY   #1       ;NEIN, SEZTE AUSTAUSCH-FLAG
        PHA            ;TAUSCHE ELEMENTE UNTER
                       ;  VERWENDUNG DES STAPELS AUS
        LDA   $41,X
        STA   $40,X
        PLA
        STA   $41,X
COUNT   DEX            ;IST DER DURCHLAUFANGESCHLOSSEN?
        BNE   PASS
        DEY            ;WAREN ALLE ELEMENTE IN DER
                       ;  RICHTIGEN REIHENFOLGE?
        BEQ   ITER     ;NEIN, MACHE EINEN WEITEREN DURCH-
                       ;  LAUF
DONE    BRK
```

Offensichtlich können wir nicht alle möglichen Eingangswerte für dieses Programm prüfen. Zwei weitere einfache Datensätze für Fehlersuch-Zwecke sind:

1) Zwei gleiche Elemente

(0040) = 02
(0041) = 00
(0042) = 00

2) Zwei Elemente, bereits in absteigender Reihenfolge

(0040) = 02
(0041) = 01
(0042) = 00

EINFÜHRUNG IN DAS TESTEN

VERWENDUNG VON TESTFÄLLEN BEI DER FEHLERSUCHE

Das Testen des Programms hängt eng mit der Fehlersuche im Programm zusammen. Sicher werden einige der Testfälle gleich sein wie die Testdaten, die für die Fehlersuche verwendet werden, wie etwa:

- **Triviale Fälle, wie keine Daten oder ein einziges Element.**
- **Spezielle Fälle, die das Programm aus irgendwelchen Gründen auswählt.**
- **Einfache Beispiele, die spezielle Teile des Programmes überprüfen.**

Im Falle des Umwandlungsprogrammes dezimal in Sieben-Segment, umfassen diese Fälle alle möglichen Situationen. Die Testdaten bestehen aus:

- Die Zahlen 0 bis 9.
- Der Grenzfall 10.
- Der willkürliche Fall 6B.

Das Programm unterscheidet keine weiteren Fälle. **Hier sind Fehlersuche und Testen effektiv das gleiche.**

Bei dem Sortierprogramm ist die Aufgabe schwieriger. Die Anzahl der Elemente könnte von 0 bis 255 reichen, und jedes der Elemente könnte irgendwo in diesem Bereich liegen. Die Zahl der möglichen Fälle ist daher außerordentlich groß. Ferner ist das Programm einigermaßen komplex. Wie sollen wir Testdaten auswählen, die uns einen gewissen Grad von Vertrauen zu diesem Programm geben? **Hier erfordert das Testen einige grundlegende Entscheidungen.** Das Testproblem ist besonders schwierig, wenn das Programm von einer Folge von Echtzeit-Daten abhängt. Wie wählen wir die Daten, erzeugen diese und führen sie dem Mikrocomputer in realistischer Weise zu?

TEST-HILFEN

Die meisten der früher erwähnten Hilfsmittel für die Fehlersuche sind auch beim Testen sehr nützlich. Logik- oder Mikroprozessor-Analysatoren können beim Überprüfen der Hardware helfen, Simulatoren beim Prüfen der Software. Andere Hilfsmittel können ebenfalls nützlich sein, zum Beispiel:

1) **E/A-Simulationen,** die eine Vielzahl von Bausteinen von einem einzelnen Eingang und einem einzelnen Ausgangs-Baustein simulieren können.
2) **"In-Circuit"-Emulatoren** gestatten Ihnen das Anschließen des Prototyps an ein Entwicklungssystem oder Steuerpult und das Testen.
3) **ROM-Simulatoren,** die die Flexibilität eines RAMs besitzen, jedoch auch die zeitliche Steuerung des speziellen ROMs und PROMs, die im endgültigen System verwendet werden.
4) **Echtzeit-Betriebssysteme,** die Eingaben oder Unterbrechungen zu speziellen Zeiten (oder vielleicht willkürlich) liefern und das Auftreten von Ausgaben markieren. Echtzeit-Haltepunkte und "Tracer" (Ablaufverfolger) können ebenso enthalten sein.
5) **Emulationen** (häufig auf mikroprogrammierbaren Computern), die eine Echtzeit-Ausführungsgeschwindigkeit und programmierbare E/A ergeben.[4]
6) **Interfaces,** die einem anderen Computer die Steuerung des E/A-Systems und das Testen des Mikrocomputer-Programmes gestatten.

7) **Testprogramme,** die jede Verzweigung in einem Programm auf logische Fehler prüfen.
8) **Testprogramme,** die willkürlich Daten oder andere Verteilungen erzeugen können.

Es gibt formelle Test-Theorien, aber sie sind gewöhnlich nur für sehr kurze Programme verwendbar.

Man muß sehr sorgfältig darauf achten, daß die Testgeräte den Test selbst nicht verfälschen, indem sie die Umgebungsbedingungen verändern. Häufig können Testgeräte Eingangs- und Ausgangssignale puffern, zwischenspeichern oder verändern. Das tatsächliche System könnte dies nicht ausführen, und könnte sich daher etwas anders verhalten. Ferner kann zusätzliche Software in dem Test-Aufbau einen Teil des Speicherraumes oder Teil des Unterbrechungssystemes verwenden. Es kann auch eine Fehler-Korrektur und andere Eigenschaften besitzen, die im endgültigen System nicht vorhanden sind. Ein Software-Test muß ebenso realistisch sein wie ein Hardware-Test, da Software-Fehler ebenso kritisch wie Hardware-Fehler sein können.

Emulationen und Simulationen sind natürlich niemals völlig genau. Sie sind gewöhnlich ausreichen für das Prüfen der Logik, können jedoch selten helfen, das Interface oder die zeitliche Steuerung zu testen. Andererseits ergeben Echtzeit-Testgeräte keinen Überblick über die Programmlogik und können das Interfacing und die zeitliche Steuerung beeinflussen.

AUSWAHL VON TESTDATEN

Sehr wenige reale Programme können für sämtliche möglichen Fälle überprüft werden. Der Entwickler muß einen Satz von Beispielen wählen, die auf irgendeine Weise den gesamten Bereich der Möglichkeiten umfassen.

STRUKTURIERTES TESTEN

Testen sollte natürlich ein Teil des gesamten Entwicklungs-Vorganges sein. Entwicklung mit schrittweiser Verfeinerung (Top-down-Design) und strukturierte Programmierung beinhalten bereits das Testen als einen Teil der Entwicklung. Dies wird strukturiertes Testen genannt.[6] Jedes Modul innerhalb eines strukturierten Programmes sollte seperat geprüft werden. **Testen sollte ebenso wie das Codieren modular, strukturiert und "Top-down" sein.**

TESTEN SPEZIELLER FÄLLE

Aber das läßt noch immer die Frage für die Auswahl von Testdaten für ein Modul offen. Der Entwickler muß zuerst alle speziellen Fälle erfassen, die ein Programm erkennt. Diese können enthalten:

- Triviale Fälle
- Gleichheits-Fälle
- Spezielle Situationen

Die Testdaten sollten alle diese Fälle umfassen.

BILDEN VON DATEN-KLASSEN

Als nächstes müssen Sie jede Klasse von Daten festlegen, die Anweisungen innerhalb des Programmes unterscheiden können. Diese können enthalten:

- Positive oder negative Zahlen
- Zahlen oberhalb oder unterhalb einer bestimmten Schwelle
- Daten, die spezielle Sequenzen oder Zeichen enthalten, oder auch nicht.
- Daten, die zu einem bestimmten Zeitpunkt nicht vorliegen.

Wenn die Module kurz sind, sollte die Gesamtzahl der Klassen noch klein sein, obwohl jede Unterteilung multiplikativ ist, das heißt, zwei Zwei-Weg-Unterteilungen ergeben vier Datenklassen.

AUSWAHL DER DATEN VON DEN KLASSEN

Sie müssen nun die Klassen entsprechend aufteilen, ob das Programm ein unterschiedliches Resultat für jede Eingabe in der Klasse (wie in einer Tabelle) erzeugt oder das gleiche Resultat für jede Eingabe (wie etwa das Warnen, daß ein Parameter sich oberhalb einer Schwelle befindet) ergibt. In einem bestimmten Fall kann man jedes Element einschließen, wenn die Gesamtzahl klein ist oder nur ein Beispiel, wenn die Zahl groß ist. Das Beispiel sollte alle Grenzfälle beinhalten und wenigstens einen willkürlich gewählten Fall. Tabellen für Zufallszahlen sind aus Büchern zu entnehmen und Generatoren für Zufallszahlen sind Teil der meisten Computer.

Sie müssen sehr sorgfältig Unterscheidungen beobachten, die nicht offensichtlich sein können. Beispielsweise wird der 6502 eine 8-Bit-Zahl ohne Vorzeichen größer als 127 als negativ ansehen. Sie müssen dies in Betracht ziehen, wenn Sie die Sprungbefehle, die von Vorzeichenbit abhängen, verwenden. Sie müssen auch auf Befehle achten, die die Flags und den Überlauf bei der Arithmetik mit Vorzeichen nicht beeinflussen, sowie zwischen Größen für die Adressenlänge (16-Bit) und Größen für Datenlängen (8-Bit) unterscheiden.

Testbeispiel 1: Sortierprogramm

TESTEN EINES SORTIERPROGRAMMS

Die speziellen Fälle sind hier offensichtlich:

- Keine Elemente in der Anordnung
- Ein Element, dessen Größe willkürlich gewählt werden kann.

Der andere spezielle, zu berücksichtigende Fall liegt vor, wenn die Elemente gleich sind.

Es kann hier einige Probleme mit Vorzeichen und Datenlängen geben. Beachten Sie, daß die Anordnung selbst weniger als 256 Elemente enthalten muß. Die Verwendung des Befehls LDY #1 anstatt INY zum Setzen des Austausch-Flags bedeutet, daß es keine Schwierigkeit geben wird, wenn die Anzahl der Elemente oder Austausch-Vorgänge 128 überschreitet. Wir könnten den Einfluß des Vorzeichens prüfen, indem wir für die Hälfte der Testfälle die Anzahl der Elemente zwischen 128 und 255 nehmen und eine andere Hälfte zwischen 2 und 127. Alle Größen sollten willkürlich gewählt werden, um die Prüfung möglichst vorurteilsfrei auszuführen.

Testbeispiel 2: Selbstprüfende Zahlen (siehe Kapitel 8)

TESTEN EINES ARITHMETISCHEN PROGRAMMS

Hier wollen wir voraussetzen, daß eine vorhergehende Gültigkeitsprüfung gesichert hat, daß die Zahl die richtige Länge besitzt und aus gültigen Ziffern besteht. Da das Programm keine weiteren Unterscheidungen macht, sollten die Testdaten willkürlich gewählt werden. Hier ist eine Tabelle mit Zufallszahlen oder ein Zufallszahlen-Generator ideal. Der Bereich der Zufallszahlen liegt zwischen 0 und 9.

VORSICHTSMASSNAHMEN BEIM TESTEN

REGELN FÜR DAS TESTEN

Der Entwickler kann die Stufe des Testens vereinfachen, indem er die Programme entsprechend entwickelt. Sie sollten folgende Regeln beachten:

1) Versuchen Sie, triviale Fälle so früh wie möglich zu eliminieren, ohne unnotwendige Unterscheidungen einzuführen.
2) Versuchen Sie, die Anzahl der speziellen Fälle auf ein Minimum zu reduzieren. Jeder spezielle Fall bedeutet zusätzliches Testen und Zeit für die Fehlersuche.
3) Überlegen Sie die Ausführung von Gültigkeits- oder Fehlerprüfung an den Daten vor deren Verarbeitung.
4) Achten Sie sorgfältig auf unbeabsichtigte oder unnötige Unterscheidungen, speziell bei der Handhabung von Zahlen mit Vorzeichen oder bei der Verwendung von Operationen, die sich auf Zahlen mit Vorzeichen beziehen.
5) Prüfen Sie Grenzfälle manuell. Sie sind sehr häufig eine Fehlerquelle. Vergewissern Sie sich, daß die Aufgaben-Definition spezifiziert, was in diesen Fällen zu geschehen hat.
6) Machen Sie das Programm soweit als möglich allgemein gültig. Jede Unterscheidung und seperate Routine erhöht die Anforderung an das Testen.
7) Unterteilen Sie das Programm und entwickeln Sie die Module so, daß das Testen in Schritten erfolgen kann, in Verbindung mit den anderen Stufen der Software-Entwicklung.

SCHLUSSFOLGERUNGEN

Fehlersuche und Testen sind die Stiefkinder des Software-Entwicklungsvorganges. Die meisten Projekte lassen für sie viel zu wenig Zeit und die meisten Bücher vernachlässigen sie. Jedoch Entwickler und Manager finden häufig, daß diese Stufen die aufwendigsten und zeitraubendsten sind. Der Fortschritt ist sehr schwierig zu messen und zu erzielen. Fehlersuche und Testen bei Mikroprozessor-Software ist verhältnismäßig schwierig, da die leistungsfähigen Hardware- und Software-Hilfsmittel, die auf größeren Computern verwendet werden können, selten für Mikrocomputer verfügbar sind.

Der Entwickler sollte Fehlersuche und Testen sorgfältig planen. Wir empfehlen folgendes Vorgehen:

1) **Versuchen Sie Programme zu schreiben, die leicht zu testen und fehlerfrei zu machen sind. Modulare Programmierung, strukturierte Programmierung und Entwicklung mit stufenweiser Verfeinerung sind sehr nützliche Techniken.**
2) **Bereiten Sie einen Fehlersuch- und Testplan als Teil der Programm-Entwicklung vor. Entscheiden Sie früh genug, welche Daten Sie erzeugen müssen und welche Geräte Sie benötigen.**
3) **Testen und machen sie jedes Modul fehlerfrei, als Teil der Entwicklung mit schrittweiser Verfeinerung.**
4) **Machen Sie die Logik jedes Moduls systematisch fehlerfrei. Verwenden Sie Prüflisten, Haltepunkte und das Einzelschritt-Verfahren. Wenn die Programmlogik komplex ist, überlegen Sie die Verwenudng des Software-Simulators.**
5) **Prüfen Sie die zeitliche Steuerung jedes Moduls systematisch, wenn dies ein Problem darstellt. Ein Ozillograf kann zahlreiche Probleme lösen, wenn Sie den Test entsprechend vorsehen. Ist die zeitliche Steuerung sehr komplex, überlegen Sie die Verwendung eines Logik- oder Mikroprozessor-Analysators.**
6) **Vergewissern Sie sich, daß die Testdaten auch repräsentative Beispiele sind. Achten Sie auf alle Klassen von Daten, die das Programm unterscheiden kann. Schließen Sie alle speziellen und trivialen Fälle ein.**
7) **Wenn das Programm jedes Element anders behandelt, oder die Anzahl der Fälle groß ist, wählen Sie die Testdaten willkürlich aus.**[7]
8) **Zeichnen Sie alle Testergebnisse als Teil der Dokumentation auf. Wenn Probleme auftreten, so brauchen Sie nicht die Testfälle zu wiederholen, die Sie bereits ausgeführt haben.**

LITERATUR

1) Für weitere Informationen über Logik-Analysatoren siehe:

N. Andreiev, "Special Report Troubleshooting Instruments," EDN, October 5, 1978, pp. 89-99

R. L. Down, "Understanding Logic Analyzers," Computer Design June 1977, pp. 188-191.

R. Gasperini, "A Guide to Digital Troubleshooting Aids," Instruments and Control Systems, February 1978, pp. 39-42.

R. Lorentzen, "Troubleshooting Microprocessors with an Logic Analyzer System," Computer Design, March 1979, pp. 160-164 (includes a 6502-based example).

M. Marshall, "What to Look for in Logic Timing Analyzer," Electronics, March 29, 1979, pp. 109-114.

K. Pines, "What Do Logic Analyzers Do?", Digital Design, September 1977, pp. 55-72.

I. Spector, "Logic Analysis by Telephone," EDN, March 20, 1979, pp. 139-142.

2) W. J. Weller, Assembly Level Programming for Small Computers, Lexington Books, Lexington, MA, 1975, Chapter 23.

3) R. L. Baldridge, "Interrupts Add Power, Complexity to Microcomputer System Design," EDN, August 5, 1977, pp. 67-73.

4) H. R. Burris, "Time-Scaled Emulations of the 8080 Microprocessor," Proceedings of the 1977 National Computer Conference, pp. 937-946.

5) D. A. Walsh, "Structures Testing," Datamation, July 1977, pp. 111-118.

P. F. Barbuto Jr. and J. Geller, "Tools for Top-Down Testing," Datamation, October 1978, pp. 178-182.

6) R. A. DeMillo et al., "Hints on Test Data Selection: Help for the Practicing Programmer," Computer, April 1978, pp. 34-41.

W. F. Dalton, "Design Microcomputer Software," Electronics, January 19, 1978, pp. 97-101.

7) T. G. Lewis, Distribution Sampling for Computer Simulation, Lexington Books, Lexington, MA, 1975.

R. A. Mueller et al., "A Random Number Generator for Microprocessors," Simulation, April 1977, pp. 123-127.

Kapitel 15
DOKUMENTATION UND NEU-ENTWICKLUNG

Das tatsächlich funktionierende Programm ist nicht die einzige Anforderung an die Software-Entwicklung. Die entsprechende Dokumentation stellt ebenso einen wichtigen Teil der Software dar. Die Dokumentation hilft nicht nur dem Entwickler beim Testen und bei der Fehlersuche, sondern ist auch für späteren Gebrauch und Erweiterung des Programmes sehr wesentlich. Ein schlecht dokumentiertes Programm wird schwierig zu warten, neuerlich zu verwenden oder zu erweitern sein.

Gelegentlich verwendet die erste Version eines Programms zuviel Speicher oder wird zu langsam ausgeführt. Der Entwickler muß dann Wege überlegen, um dieses Programm zu verbessern. Diese Stufe wird Neu-Entwicklung genannt und erfordert, daß Sie sich auf jene Teile des Programms konzentrieren, bei denen eine Verbesserung den größten Erfolg verspricht.

SELBST-DOKUMENTIERENDE PROGRAMME

REGELN FÜR SELBST-DOKUMENTIERENDE PROGRAMME

Obwohl kein Programm jemals selbständig dokumentierend ist, können einige der früher erwähnten Regeln sehr nützlich sein. Diese beinhalten:

- Klare einfache Struktur mit so wenig Steuerungstransfers (Sprüngen) wie möglich.
- Verwendung sinnvoller Namen und Markierungen.
- Verwendung von Namen für E/A-Bausteine, Parameter, numerische Faktoren etc.
- Es sollte besonderer Wert auf Einfachheit anstatt auf geringere Ersparnisse bei der Verwendung des Speichers, Ausführungszeit etc. gelegt werden.

Beispielsweise sendet das folgende Programm eine Reihe von Zeichen zu einem Fernschreiber:

```
        LDX     $40
W       LDA     $0FFF,X
        STA     $A000
        JSR     XXX
        DEX
        BNE     W
        BRK
```

Sogar ohne Kommentare können wir dieses Programm wie folgt verbessern:

```
MESSG   =$1000
COUNT   =$40
TTYVIA  =A000
        LDX    COUNT
OUTCH   LDA    MESSG-1,X
        STA    TTYVIA
        JSR    BITDLY
        DEX
        BNE    OUTCH
        BRK
```

Sicher ist dieses Programm leichter zu verstehen als die frühere Version. Auch ohne weitere Dokumentation können Sie wahrscheinlich die Arbeitsweise des Programms begreifen sowie die Bedeutung der meisten Variablen erkennen. **Andere Dokumentationsverfahren können die Selbst-Dokumentation nicht ersetzen.**

AUSWAHL NÜTZLICHER NAMEN

Einige weitere Anmerkungen bezüglich der Auswahl von Namen:

1) **Verwenden Sie sinnvolle Namen,** wenn diese verfügbar sind, wie TTY oder CRT für Ausgangsgeräte, START oder RESET für Adressen, DELAY oder SORT für Unterprogramme, COUNT oder LENGTH für Daten.
2) **Vermeiden Sie Abkürzungen** wie S 16BA für SORT 16-BIT ARRAY. Dies wird selten etwas für den Anwender bedeuten.
3) **Verwenden Sie volle Worte** oder möglichst nahezu vollkommene Worte wenn möglich, wie DONE, PRINT, SEND etc.
4) **Halten Sie die Namen so charakteristisch wie möglich.**

KOMMENTARE

Die offensichtlichste Form zusätzlicher Dokumentation ist der Kommentar. Jedoch sehr wenige Programme (einschließlich der meisten in Büchern) besitzen wirklich effektive Kommentare. Sie sollten die folgenden Richtlinien für gute Kommentare beachten:

RICHTLINIEN FÜR KOMMENTARE

1) **Wiederholen Sie nicht die Bedeutung des Befehlscodes.** Erklären Sie eher den Zweck des Befehls im Programm. Kommentare wie

```
DEX     ;X=X-1
```

tragen überhaupt nichts zur Dokumentation bei. Verwenden Sie stattdessen

```
DEX     ;ZEILENZAHL = ZEILENZAHL-1
```

Erinnern Sie sich daran, daß Sie ja bereits wissen, was der Operationscode bedeutet und diese jeder in einem Handbuch nachschlagen kann. **Es ist viel wichtiger zu erklären, welche Aufgabe das Programm ausführt.**

2) **Machen Sie die Kommentare so deutlich wie möglich.** Verwenden Sie keine Abkürzungen oder Akronyme, außer sie sind allgemein bekannt (wie ASCII, VIA oder UART) oder Standard (wie Nr. für Nummer, ms für Millisekunden, etc.). Vermeiden Sie Kommentare wie

```
DEX          ;LN = LN-1
```

oder

```
DEX          ;DEC LN UM 1
```

Die wenigen zusätzlichen Eingaben stellen keinen nennenswerten Aufwand dar.

3) **Kommentieren Sie jeden wichtigen oder nicht sehr offensichtlichen Punkt.** Seien Sie sorgfältig beim Markieren von Operationen, die keine offensichtliche Funktion besitzen, wie etwa

```
AND   #%00100000   ;TAPE READER-BIT AUS
```

oder

```
LDA   GCODL,X      ;UMWANDLUNG IN GRAY-CODE MIT
                   ;  TABELLE
```

Offensichtlich erfordern E/A-Operationen häufig ausführliche Kommentare. Wenn Sie sich nicht ganz sicher darüber sind, was ein Befehl ausführt, oder Sie darüber nachdenken müssen, fügen Sie einen erklärenden Kommentar hinzu. Der Kommentar wird Ihnen später viel Zeit ersparen und nützlich bei der Dokumentation sein.

4) **Kommentieren Sie keine offensichtlichen Dinge.** Ein Kommentar auf jeder Zeile macht es schwierig, die wichtigen Punkte aufzufinden. Standard-Sequenzen wie

```
        DEX
        BNE     SEARCH
```

müssen nicht markiert werden, außer Sie führen irgend etwas Spezielles aus. Ein Kommentar wird häufig für mehrere Zeilen ausreichen wie in

```
LSR   A       ;HOLE DIE HÖCHSTWERTIGE STELLE
LSR   A
LSR   A
LSR   A
LSR   A
LDA   $40     ;TAUSCHE HÖCHSTWERTIGE UND
LDA   $41     ;   NIEDRIGSTWERTIGE BYTES AUS
STA   $41
STX   $40
```

5) **Plazieren Sie Kommentare auf die Zeilen, auf die sie sich beziehen, oder auf den Beginn einer Sequenz.**
6) **Halten Sie Ihre Kommentare immer auf dem neuesten Stand.** Wenn Sie ein Programm ändern, ändern Sie auch die Kommentare.
7) **Verwenden Sie Standardformen und Ausdrücke** beim Kommentieren. Stoßen Sie sich nicht an Wiederholungen. Unterschiedliche Namen für die gleichen Dinge sind verwirrend. Auch wenn die Variationen etwa COUNT und COUNTER, START und BEGINN, DISPLAY und LEDS oder PANEL und SWITCHES sind.
   Es bringt auf jeden Fall Gewinn, wenn man konsequent ist. Die Variationen können im Augenblick offensichtlich sein, jedoch später nicht mehr so klar erscheinen. Andere werden Sie von Anfang an verwirren.
8) **Halten Sie die Kommentare kurz.** Überlassen Sie eine vollständige Beschreibung der Dokumentation. Andernfalls kann das Programm in den Kommentaren untergehen, und Sie werden Mühe haben, es zu finden.
9) **Verbessern Sie ständig Ihre Kommentare.** Wenn Sie einen Kommentar nicht lesen oder verstehen können, nehmen Sie sich die Zeit, es zu ändern. Wenn Sie merken, daß die Auflistung verwirrend wird, fügen Sie einige leere Zeilen hinzu. Die Kommentare werden sich nicht selbst verbessern. In der Tat werden sie schwierig, wenn Sie eine Aufgabe erledigt haben und vergessen, was das Programm genau ausgeführt hat.
10) **Vor jedem größeren Abschnitt, Unterabschnitt oder Unterprogramm setzen Sie eine Anzahl von Kommentaren ein, die die Funktionen des folgenden Codes beschreiben.** Besonders sorgfältig sollten alle Eingaben, Ausgaben und Nebeneffekte beschrieben werden, sowie die verwendeten Algorithmen.
11) **Es bewährt sich immer, wenn beim Modifizieren von Programmen Kommentare angebracht werden, die das Datum, den Autor und die Art der ausgeführten Modifikation aufzeigen.**

Erinnern Sie sich daran, daß Kommentare wichtig sind. Gute Kommentare werden Ihnen Zeit und Mühe sparen. Investieren Sie einige Zeit in die Kommentare und versuchen Sie, diese so effektiv wie möglich zu machen.

KOMMENTIER-BEISPIEL 1: ADDITION MIT MEHRFACHER GENAUIGKEIT

BEISPIELE FÜR KOMMENTARE

Das grundlegende Programm lautet:

```
         LDX   $40
         CLC
ADDWD    LDA   $40,X
         ADC   $50,X
         STA   $40,X
         DEX
         BNE   ADDW
         BRK
```

Kommentieren Sie zuerst die wichtigen Punkte. Diese sind typisch die Initialisierungen, das Holen von Daten und die Verarbeitungs-Operationen. Befassen Sie sich nicht mit Standard-Sequenzen, wie das Weiterstellen von Zeigern und Zählern. Erinnern Sie sich daran, daß Namen deutlicher als Zahlen sind, und verwenden Sie daher diese weitgehend.

Die neue Version des Programms lautet:

```
;
;ADDITION MIT MEHRFACHER GENAUIGKEIT
;
;DIESES PROGRAMM FÜHRT EINE BINÄRE MEHRBYTE-ADDITION AUS
;
;  EINGABEN: SPEICHERPLATZ 0040(HEX) = LÄNGE DER ZAHLEN (IN BYTES)
;            SPEICHERPLÄTZE 0041 BIS 0050 (HEX) = ERSTE ZAHL
;            BEGINNEND MIT MSB'S
;            SPEICHERPLÄTZE 0051 BIS 0060 (HEX) = ZWEITE ZAHL
;            BEGINNEND MIT MSB'S
;  AUSGABEN: SPEICHERPLÄTZE 0041 BIS 0050 (HEX) = SUMME BEGINNEND
;            MIT MSB'S
;
LENGTH  =$40
NUMB1   =$41
NUMB2   =$51
        LDX   LENGTH       ;ZÄHLUNG = LÄNGE DER ZAHLEN (IN
                           ;  BYTES)
        CLC
ADDWD   LDA   NUMB1-1,X    ;HOLE BYTE VON REIHE 1
        ADC   NUMB2-1,X    ;ADDIERE BYTE VON REIHE 2
        STA   NUMB1-1,X    ;SPEICHERE ERGEBNIS IN REIHE 1
        DEX
        BNE   ADDWD        ;SETZE FORT BIS ALLE BYTES ADDIERT
                           ;  SIND
        BRK
```

Als zweites sehen Sie sich jeden Befehl an, der keine offensichtliche Funktion haben könnte und markieren Sie diese. Hier ist der Zweck von CLC das Löschen des Übertrags beim ersten Durchlauf.

FRAGEN BEI DER KOMMENTIERUNG

Drittens fragen Sie sich selbst, ob die Kommentare Ihnen das sagen, was Sie bei der Verwendung des Programms wissen wollen, zum Beispiel:

1) Wo wird in das Programm eingetreten? Gibt es alternative Eintritts-Punkte?
2) Welche Parameter sind erforderlich? Wie und in welcher Form müssen sie geliefert werden?
3) Welche Operationen führen das Programm aus?
4) Von wo werden die Daten geholt?
5) Wo werden die Ergebnisse gespeichert?
6) Welche speziellen Fälle werden berücksichtigt?
7) Was macht das Programm bei Fehlern?
8) Wie wird aus dem Programm ausgetreten?

Einige der Fragen können vielleicht für ein spezielles Programm nicht relevant sein und einige der Antworten vielleicht offensichtlich. Vergewissern Sie sich, daß Sie sich nicht extra hinsetzen und das Programm zerlegen müssen um herauszufinden, wie die Antworten lauten. Erinnern Sie sich daran, daß zuviel Erklärung unnötiger Ballast ist, den Sie erst wieder beseitigen müssen. Gibt es irgend etwas, was Sie zu dieser Liste hinzufügen oder entfernen würden? Wenn ja, führen Sie es aus – Sie sind dann jemand, der ein Gefühl dafür hat, daß das Kommentieren ausreichend und vernünftig ist.

```
;
;ADDITION MIT MEHRFACHER GENAUIGKEIT
;
;
;DIESES PROGRAMM FÜHRT EINE BINÄRE MEHRBYTE-ADDITION AUS
;
;  EINGABEN: SPEICHERPLATZ 0040 (HEX) = LÄNGE DER ZAHLEN (IN
;                 BYTES)
;                 SPEICHERPLÄTZE 0041 BIS 0050 (HEX) = ERSTE ZAHL
;                 BEGINNEND MIT MSB'S
;                 SPEICHERPLÄTZE 0051 BIS 0060 (HEX) = ZWEITE ZAHL
;                 BEGINNEND MIT MSB'S
;  AUSGABEN: SPEICHERPLÄTZE 0041 BIS 0050 (HEX) = SUMME BEGINNEND
;                 MIT MSB'S
;
LENGTH  =$40                  ;LÄNGE DER ZAHLEN (IN BYTES)
NUMB1   =$41                  ;MSB'S DER ERSTEN ZAHL UND ERGEBNIS
NUMB2   =$51                  ;MSB'S DER ZWEITEN ZAHL
        LDX   LENGTH          ;ZÄHLUNG = LÄNGE DER ZAHLEN (IN
                              ;  BYTES)
        CLC                   ;LÖSCHE ÜBERTRAG ZU BEGINN
ADDWD   LDA   NUMB1-1,X       ;HOLE BYTE VON REIHE 1
        ADC   NUMB2-1,X       ;ADDIERE BYTE VON REIHE 2
        STA   NUMB1-1,X       ;SPEICHERE ERGEBNIS IN REIHE 1
        DEX
        BNE   ADDWD           ;SETZE FORT BIS ALLE BYTES ADDIERT
                              ;  SIND
        BRK
```


Kommentier-Beispiel 2: Fernschreiber-Ausgabe

Das grundlegende Programm lautet:

```
        LDA   $60
        ASL   A
        LDX   #11
TBIT    STA   $A000
        JSR   BITDLY
        ROR   A
        SEC
        DEX
        BNE   TBIT
        BRK
```

Das Kommentieren der wichtigsten Punkte und Hinzufügen von Namen ergibt:

```
;
;FERNSCHREIBER-AUSGABE
;
;DIESES PROGRAMM DRUCKT DEN INHALT DER SPEICHERPLÄTZE 0060
;  (HEX) ZUM FERNSCHREIBER
;
;  EINGABEN: ZU SENDENDE ZEICHEN IM SPEICHERPLATZ 0060
;  AUSGABEN: KEINE
;
NBITS   =11                   ;ANZAHL DER BITS JE ZEICHEN
TDATA   =$60                  ;ADRESSE DES ZU SENDENDEN ZEICHENS
TTYVIA  =$A000                ;FERNSCHREIBERAUSGANGS-DATENPORT
        LDA   $60             ;HOLE DATEN
        ASL   A               ;VERSCHIEBE NACH LINKS UND BILDE
                              ;  STARTBIT
        LDX   #NBITS          ;ZÄHLUNG = ANZAHL DER BITS IN ZEICHEN
TBIT    STA   TTYVIA          ;SENDE NÄCHSTES BIT ZUM
                              ;  FERNSCHREIBER
        JSR   BITDLY          ;WARTE 1 BITZEIT
        ROR   A               ;HOLE NÄCHSTES BIT
        SEC                   ;SETZE ÜBERTRAG ZUM BILDEN DER
                              ;  STOPBITS
        DEX
        BNE   TBIT            ;ZÄHLE BITS
        BRK
```

Beachten Sie, wie leicht wir dieses Programm ändern könnten, so daß es eine ganze Reihe von Daten transferieren würde, beginnend bei der Adresse in den Speicherplätzen DPTR und DPTR+1 und endend mit einem "03"- (ASCII ETX-) Zeichen. Ferner wollen wir das Terminal zu einem Gerät mit 30 cps und mit einem Stopbit machen (wir werden das Unterprogramm BITDLY zu ändern haben). Versuchen Sie die Änderungen auszuführen, bevor Sie sich nachstehende Auflistung ansehen:

```
;
;AUSGABEPROGRAMM FÜR EINE ZEICHENREIHE
;
;DIESES PROGRAMM SENDET EINE REIHE VON ZEICHEN ZU EINEM
;  TERMINAL MIT 30 CPS. ÜBERTRAGUNG WIRD BEENDET, WENN EIN
;  ASCII-ZEICHEN ETX (03 HEX) FESTGESTELLT WIRD
;
;  EINGABEN: SPEICHERPLÄTZE 0060 UND 0061 (HEX) ENTHALTEN
;            ADRESSE DER ZU SENDENDEN REIHE
;  AUSGABEN: KEINE
;
DPTR    =$60                   ;ZEIGER ZUM AUSGANGS-DATENPUFFER
ENDCH   =$03                   ;ABSCHLUSSZEICHEN = ASCII ETX
NBITS   =10                    ;ANZAHL DER BITS PRO ZEICHEN
TRMVIA  =$A000                 ;AUSGANGS-DATENPORT DES TERMINALS
        LDY    #0              ;ZEIGE ZUM BEGINN DES AUSGANGS-
                               ;  DATENPUFFERS
TCHAR   LDA    (DPTR),Y        ;HOLE EIN ZEICHEN VOM PUFFER
        CMP    #ENDCH          ;IST ES EIN ABSCHLUSSZEICHEN?
        BEQ    DONE;           JA, DONE
        ASL    A               ;NEIN, VERSCHIEBE NACH LINKS UND
                               ;  BILDE STARTBIT
        LDX    NBITS           ;ZÄHLUNG = ANZAHL DER BITS JE ZEICHEN
TBIT    STA    TRMVIA          ;SENDE NÄCHSTES BIT ZUM TERMINAL
        JSR    BITDLY          ;WARTE 1 BITZEIT
        ROR    A               ;HOLE NÄCHSTES BIT
        SEC                    ;SETZE ÜBERTRAG ZUM BILDEN DES
                               ;  STOPBITS
        DEX
        BNE    TBIT            ;ZÄHLE BITS
        INY                    ;GEHE ZUM NÄCHSTEN ZEICHEN WEITER
        JMP    TCHAR
```

Gute Kommentare erleichtern Ihnen wesentlich die Änderung eines Programms, falls neue Anforderungen auftreten. Versuchen Sie beispielsweise, das letzte Programm so zu ändern, daß es:

- jede nachricht mit ASCII STX (02_{16}) beginnt, gefolgt von einem dreistelligen Identifikationscode, gespeichert in den Speicherplätzen IDCODE bis IDCODE+2,
- keine Start- oder Stopbits hinzufügt,
- eine Millisekunde zwischen den Bits wartet,
- 40 Zeichen sendet, beginnend mit dem einen, das in der Adresse DPTR und DPTR+1 liegt,
- jede Mitteilung mit zwei aufeinanderfolgenden ASCII ETX (03_{16}) abschließt.

FLUSSDIAGRAMME ALS DOKUMENTATION

GRÜNDE FÜR DIE VERWENDUNG VON FLUSSDIAGRAMMEN

Wir haben bereits die Verwendung von Flußdiagrammen als Hilfsmittel bei der Entwicklung in Kapitel 13 beschrieben. Flußdiagramme sind ebenfalls bei der Dokumentation nützlich, speziell wenn:

- sie nicht zu sehr detailliert und dadurch schlecht lesbar sind,
- ihre Entscheidungspunkte deutlich erklärt und markiert sind,
- sie alle Verzweigungen beinhalten,
- sie mit der tatsächlichen Programm-Auflistung übereinstimmen.

Flußdiagramme sind nützlich, wenn sie Ihnen einen allgemeinen Überblick über das Programm geben. Sie sind von geringem Nutzen, wenn sie ebenso schwierig wie eine gewöhnliche Auflistung zu lesen sind.

STRUKTURIERTE PROGRAMME ALS DOKUMENTATION

Strukturierte Programme (siehe Kapitel 13) können auch als Teil der Dokumentation dienen, wenn:

- Sie den Zweck jedes Abschnittes in den Kommentaren beschreiben,
- Sie es mit Anweisungen verdeutlichen, die jede bedingte oder Schleifen-Struktur beinhaltet, indem Identifizierungs- und Abschlußmarkierungen verwendet werden,
- Sie die gesamte Struktur so einfach wie möglich machen,
- Sie eine konsequente, gut definierte Sprache verwenden.

Das strukturierte Programm kann Ihnen helfen, die Logik zu prüfen oder sie zu verbessern. Ferner, da das strukturierte Programm maschinen-unabhängig ist, kann es Ihnen auch bei der Ausführung einiger Aufgaben auf anderen Computern helfen.

SPEICHERPLÄNE

Ein Speicherplan ist einfach eine Auflistung aller Speicherzuweisungen in einem Programm. Der Plan gestattet Ihnen zu bestimmen, wieviel Speicher benötigt wird, die Lage der Daten oder Unterprogramme und die Teile des Speichers, die noch nicht zugewiesen wurden. Der Plan ist eine sehr handliche Referenz für das Auffinden von Speicherplätzen und Eingabepunkten und für die Aufteilung des Speichers zwischen verschiedenen Routinen oder Programmierern. Der Plan wird Ihnen auch einen einfachen Zugriff auf Daten und Unterprogramme geben, wenn Sie diese bei späteren Erweiterungen oder bei der Betreuung des Programmes benötigen. Manchmal ist ein grafischer Plan nützlicher als eine Auflistung.

Ein typischer Speicherplan wird folgendermaßen aussehen:

TYPISCHER SPEICHER-PLAN

Programmspeicher		
Adresse	Routine	Zweck
E000-E1FF	INTRPT	Unterbrechungs-Service-Routine für Tastatur
E200-E240	BRKPT	Service-Routine für Break-Befehl
E241-E250	DELAY	Verzögerungsprogramm
E251-E270	DSPLY	Anzeige-Steuerprogramm
E271-E3F9	MAIN	Hauptprogramm
E3FA-E3FF		Unterbrechungs- und Reset-Vektoren
Datenspeicher		
0000	NKEYS	Anzahl der Tasten
0001-0002	KPTR	Tastatur-Puffer-Zeiger
0003-0041	KBFR	Tastatur-Puffer
0042-0051	DBFR	Anzeige-Puffer
0051-006F	TEMP	Zeitweilige Speicherung
0100-01FF	STACK	RAM-Stapel

Erinnern Sie sich daran, daß der RAM-Stapel des 6502 immer auf Seite 1 des Speichers liegt.

PARAMETER- UND DEFINITIONSLISTEN

Parameter- und Definitionslisten am Beginn des Programms und jedes Unterprogramms machen das Verständnis und Änderungen des Programms wesentlich einfacher. Die folgenden Regeln können helfen:

REGELN FÜR DEFINITIONSLISTEN

1) **Trennen sie RAM-Speicherplätze, E/A-Einheiten, Parameter, Definitionen und Speichersystem-Konstanten.**
2) **Ordnen Sie Listen, wenn möglich, alphabetisch an** – mit einer Beschreibung jeder Eingabe.
3) **Geben Sie jedem Parameter, der sich ändern könnte, einen Namen und nehmen Sie diesen in die Liste auf.** Derartige Parameter können Zeitkonstanten, Eingaben oder Codes entsprechend spezieller Tasten oder Funktionen, Steuer- oder Maskier-Muster, Start- oder Abschlußzeichen, Schwellen etc. enthalten.
4) **Führen Sie die Speichersystem-Konstanten in einer separaten Liste auf.** Diese Konstanten werden die Reset- und Unterbrechungs-Service-Adressen enthalten, die Start-Adresse des Programms, RAM-Bereiche, Stapel-Bereiche etc.
5) **Geben Sie jedem Port, der von einem E/A-Baustein verwendet wird, einen Namen,** auch wenn Bausteine sich den gleichen Port in dem momentanen System teilen würden. Diese Trennung macht eine Erweiterung oder Neu-Anordnung wesentlich einfacher.

Eine typische Liste von Definitionen wäre:

TYPISCHE DEFINTIONS-LISTE

```
;
;SPEICHERSYSTEM-KONSTANTE
;
INTRP    =$E200          ;UNTERBRECHUNGS-EINTRITTS-PUNKT
RAMST    =$0             ;BEGINN DES DATENSPEICHERS
RESET    =$E300          ;RESET-ADRESSE
STPTR    =$01FF          ;BEGINN DES STAPELS IM RAM (AUF SEITE 1)
;
;E/A-EINHEITEN
;
DSPLY    =$A000          ;AUSGABE-VIA FÜR ANZEIGEN
KBDIN    =$A001          ;EINGABE-VIA FÜR TASTATUR
KBDOT    =$A000          ;AUSGABE-VIA FÜR TASTATUR
TTYVIA   =$A800          TTY-DATENPORT
;
;RAM-SPEICHER
;
         *=RAMST
NKEYS    *=*+1           ;ANZAHL DER TASTEN
KPTR     *=*+2           ;TASTATUR-PUFFER-ZEIGER
KBFR     *=*+$40         ;TASTATUR-EINGABE-PUFFER
DBFR     *=*+$10         ;ANZEIGE-DATENPUFFER
TEMP     *=*+$14         ;ZEITWEILIGE SPEICHERUNG
```

```
;
;PARAMETER
;
BOUNCE  =2        ;ENTPRELL-ZEIT IN MS
GOKEY   =10       ;IDENTIFIZIERUNG DER "GO"-TASTE
MSCNT   =$C7      ;ZÄHLUNG FÜR 1 MS-VERZÖGERUNG
OPEN    =$0F      ;MUSTER FÜR OFFENE TASTEN
TPULS   =1        ;IMPULSLÄNGE FÜR ANZEIGEN
;
;DEFINITIONEN
;
ALLHI   =$FF      ;MUSTER AUS LAUTER EINSEN
STCON   =$08      ;MUSTER FÜR BEGINN DES UMWANDLUNGS-
                  ;  IMPULSES
```

Natürlich werden die RAM-Eingaben nicht in alphabetischer Reihenfolge vorliegen, da der Entwickler diese so anordnen muß, daß er die Anzahl der Adressen-Änderungen, die in dem Programm erforderlich sind, auf ein Minimum hält.

BIBLIOTHEKS-ROUTINEN

Eine Standard-Dokumentation von Unterprogrammen wird Ihnen den Aufbau einer Bibliothek aus nützlichen Programmen ermöglichen. Der Gedanke besteht darin, auf diese Programme so leicht wie möglich zugreifen zu können. Ein Standard-Format wird Ihnen oder sonst irgendjemandem gestatten, auf einen Blick zu sehen, was das Programm ausführt. Das beste Verfahren besteht in der Schaffung einer Standardform und deren konsequente Verwendung. Bewahren Sie diese Programme gut organisiert auf (zum Beispiel entsprechend dem Prozessor, Sprache und Arten des Programms), und Sie werden bald eine sehr nützliche Sammlung besitzen. **Aber erinnern Sie sich daran, daß ohne entsprechende Organisation und gute Dokumentation die Verwendung der Bibliothek schwieriger sein kann, als das Programm jeweils neu zu schreiben.** Das Beseitigen von Fehlern in einem System erfordert ein genaues Verständnis aller Effekte jedes Unterprogramms.

STANDARD-FORMEN FÜR PROGRAMM-BIBLIOTHEK

Informationen, die Sie in der Standardform benötigen werden:

- Zweck des Programms
- Verwendeter Prozessor
- Verwendete Sprache
- Erforderliche Parameter und wie sie zum Unterprogramm übertragen werden.
- Erzeugte Ergebnisse und wie sie zum Hauptprogramm übertragen werden.
- Anzahl der verwendeten Speicherbytes
- Anzahl der erforderlichen Taktzyklen. Diese Zahl kann ein Durchschnittswert oder ein typischer Wert sein oder auch im weitem Maße variieren. Die tatsächliche Ausführungszeit wird natürlich von der Taktrate des Prozessors abhängen.
- Beeinflußte Register
- Beeinflußte Flags
- Ein typisches Beispiel
- Handhabung von Fehlern
- Spezielle Fälle
- Dokumentierte Programm-Auflistung

Wenn das Programm komplex ist, sollte die Standard-Bibliotheksform auch ein allgemeines Flußdiagramm oder ein strukturiertes Programm enthalten. Wie wir früher erwähnt haben, ist ein Bibliotheksprogramm wahrscheinlich dann am nützlichsten, wenn es eine einzelne bestimmte Aufgabe in verhältnismäßig allgemeiner Weise ausführt.

BIBLIOTHEKS-BEISPIELE

Bibliotheks-Beispiel 1: Summe von Daten

Zweck: Das Programm SUM8 berechnet die Summe eines Satzes von Binärzahlen mit 8 Bits und ohne Vorzeichen.

Sprache: 6502-Assembler.

Anfangsbedingungen: Adresse eins weniger als die Startadresse des Zahlensatzes in den Speicherplätzen 0040 und 0041, Länge des Satzes im Index-Register Y.

Abschluß-Bedingungen: Summe in Akkumulator.

Erfordernisse:

Speicher – 9 Bytes
Zeit – 7+12n Taktzyklen, wobei n die Länge des Satzes der Zahlen ist. Kann länger sein, wenn Seitengrenzen überquert werden.
Register – A,Y
RAM – Speicherplätze 0040 und 0041
Alle Flags beeinflußt

Typischer Fall: (Alle Daten in hexadezimal)

```
Start:
(0040 und 0041) = 004F
              Y = 03
         (0050) = 27
         (0051) = 3E
         (0052) = 26
Ende:
              A = 8B
```

Handhabung der Fehler: Das Programm ignoriert sämtliche Überträge. Das Übertrags-Bit gibt nur die letzte Operation wieder. Anfangsbedingungen des Indexregisters Y müssen 1 oder mehr sein. Dezimal-Betriebsart-Flag sollte gelöscht werden.

Auflistung:

```
;SUMME VON 8-BIT-DATEN
;
SUM8   LDA   #0        ;SUMME = NULL
ADD8   CLC             ;ÜBERTRAG LÖSCHEN JEDES MAL
       ADC   ($40),Y   ;SUMME = SUMME + DATENEINGABE
       DEY
       BNE   ADD8
       RTS
```


Bibliotheks-Beispiel 2: Dezimal-in-Sieben-Segment-Umwandlung

Zweck: Das Programm SEVEN wandelt eine Dezimalzahl in einen Sieben-Segment-Anzeigecode um.

Sprache: 6502-Assembler.

Anfangsbedingungen: Daten im Index-Register X.

Abschlußbedingungen: Sieben-Segment-Code im Akkumulator.

Erfordernisse:

Speicher – 19 Bytes, einschließlich der Sieben-Segment-Code-Tabelle (10 Eingaben).
Zeit – 16 Taktzyklen, wenn dezimal, 13 wenn nicht. Kann länger sein, wenn Seitengrenzen überschritten werden.
Register – A,X
Alle Flags beeinflußt.

Eingangsdaten im Index-Register X bleiben unverändert.

Typischer Fall: (Daten in hexadezimal)

```
Start
        X = 05
End:
        A = 6D
```

Fehlerhandhabung: Das Programm gibt Null in den Akkumulator zurück, wenn die Daten keine Dezimalziffern sind.

Auflistung:

```
;
;DEZIMAL-IN-SIEBEN-SEGMENT-UMWANDLUNG
SEVEN  LDA   #0         ;HOLE FEHLERCODE ZUM AUSTASTEN DER
                        ;  ANZEIGE
       CPX   #10        ;SIND DIE DATEN DEZIMAL?
       BCS   DONE       ;NEIN, BEWAHRE FEHLERCODE AUF
       LDA   #$SEG,X    ;JA, HOLE SIEBEN-SEGMENTCODE VON
                        ;  TABELLE
DONE   RTS
SSEG   .BYTE $3F,$06,$5B,$4F,$66
       .BYTE $6D,$7D,$07,$7F,$6F
```

Bibliotheks-Beispiel 3: Dezimale Summe

Zweck: Das Programm DECSUM addiert zwei Mehrwort-Dezimalzahlen.

Sprache: 6502-Assembler.

Anfangsbedingungen: Die Adresse der MSBs der einen Zahl liegt in den Speicherplätzen 0040 und 0041, die Adresse der MSBs der anderen Zahl in den Speicherplätzen 0042 und 0043. Die Länge der Zahlen (in Bytes) liegt im Index-Register Y. Die angeordneten Zahlen beginnen mit den höchstwertigen Stellen.

Abschlußbedingungen: Die Summe ersetzt die Zahl mit der Start-Adresse in den Speicherplätzen 0040 und 0041.

Erfordernisse:

Speicher – 14 Bytes
Zeit – 11+22n Taktzyklen, wobei n die Anzahl der Bytes ist. Kann länger sein, wenn Seitengrenzen überschritten werden.
Register – A,Y
RAM – Speicherplätze 0040 bis 0043
Alle Flags beeinflußt – Übertrag zeigt, wenn die Summe einen Übertrag erzeugt. Dezimal-Betriebsart-Flag wird gelöscht.

Typischer Fall: (Alle Daten in hexadezimal)

Start:
(0040 und 0041) = 0060
(0042 und 0043) = 0050
(Y) = 02
(0060) = 55
(0061) = 34
(0050) = 15
(0051) = 88
Ende:
(0060) = 71
(0061) = 22
Übertrag = 0

Fehlerhandhabung: Das Programm überprüft nicht die Gültigkeit von dezimalen Eingaben. Der Inhalt des Index-Registers Y muß 1 oder größer sein.

Auflistung:

```
DECSUM SED                ;MACHE GESAMTE ARITHMETIK DEZIMAL
       CLC                ;LÖSCHE ÜBERTRAG ZU BEGINN
DECADD DEY
       LDA    ($40),Y     ;HOLE 2 DEZIMALE STELLEN VON REIHE 1
       ADC    ($42),Y     ;ADDIERE 2 STELLEN VON REIHE 2
       STA    ($40),Y     ;SPEICHERE ERGEBNIS IN REIHE 1
       TYA
       BNE    DECADD
       CLD                ;KEHRE ZU BINÄRER BETRIEBSART
                          ;  ZURÜCK
       RTS
```


VOLLSTÄNDIGE DOKUMENTATION

DOKUMENTATIONS-PAKET

Die vollständige Dokumenation von Mikroprozessor-Software wird alle oder die meisten Elemente enthalten, die wir vorher erwähnt haben.
So kann das vollständige Dokumentations-Paket beinhalten:

- **Allgemeine Flußdiagramme**
- **Eine geschriebene Beschreibung des Programms**
- **Eine Liste aller Parameter und Definitionen**
- **Ein Speicherplan**
- **Eine dokumentierte Auflistung des Programms**
- **Eine Beschreibung des Testplans und der Test-Ergebnisse**

Die Dokumentation kann auch beinhalten:

- **Flußdiagramme des Programmierers**
- **Strukturierte Programme**

Die oben aufgeführten Dokumentationsverfahren sind ein Minimum an Dokumenten für Nicht-Produktions-Software. Produktions-Software benötigt weit größere Anstrengungen bezüglich der Dokumentation. Die folgenden Dokumente sollten ebenfalls hergestellt werden:

- Handbuch für die Programmlogik
- Anwender-Handbuch
- Service-Handbuch

Das Handbuch für die Programmlogik erweitert die geschriebenen Erläuterungen bei der Software. Es sollte für einen technisch kompetenten Anwender geschrieben werden, der möglicherweise nicht die detaillierten Kenntnisse besitzt, die bei der Beschreibung der Software vorausgesetzt wird. Das Handbuch für die Programmlogik sollt die wichtigsten Ziele des Systems beschreiben, die verwendeten Algorithmen und welche Lösungen hierzu erforderlich waren.

Es sollte dann ausführlich die verwendeten Datenstrukturen beschreiben und zeigen wie diese gehandhabt werden. Es sollte ferner einen Führer darstellen, der schrittweise durch die Operationen und das Programm leitet. Schließlich sollte es alle speziellen Tabellen oder Graphen beinhalten, die bei der Erklärung des Programms behilflich sein können. Code-Umwandlungstabellen, Zustands-Diagramme, Übersetzungs-Matrizen und Flußdiagramme sollten ebenfalls darin enthalten sein.

Das Handbuch für den Anwender ist wahrscheinlich das wichtigste und wird trotzdem am häufigsten in der Dokumentation übersehen. Unabhängig davon, wie gut die Entwicklung eines Systems ist, hat es keinen Wert, wenn niemand die Vorteile dieses Handbuchs voll nützen kann. Das Handbuch für den Anwender sollte in das System für alle Anwender einführen und möglichst verständlich gehalten sein. Es sollte dann eine detaillierte Erklärung der Eigenschaften des Systems und dessen Anwendung liefern. Zahlreiche Beispiele werden zur Erklärung der Punkte im Text dienlich sein. Eine schrittweise Einführung durch das System sollte ebenfalls vorgesehen (und getestet) sein. Programmierer mit detaillierten Kenntnissen des Systems verwenden häufig Abkürzungen, die ein Leser im allgemeinen nicht verstehen kann. Eine weitere Besprechung über das Schreiben von Anwender-Handbüchern geht über den Rahmen dieses Buches hinaus. Erinnern Sie sich jedoch daran, daß Sie niemals zuviel Mühe bei der Vorbereitung eines derartigen Handbuches aufwenden können, da es sicherlich das Dokument sein wird, auf das man sich am häufigsten bezieht.

Das Service-Handbuch ist für den Programmierer, der das System zu modifizieren hat. Es sollte ausführliche Verfahren, Schritt-für-Schritt, für derartige Rekonfigurationen beinhalten. Zusätzlich sollte es alle Vorkehrungen ausführlich beschreiben, die in dem Programm für zukünftige Erweiterungen vorgesehen sind.

Die Dokumentation sollte nicht zu leicht genommen werden oder bis zum Ende der Software-Entwicklung hinausgeschoben werden. Ordnungsmäßige Dokumentation, kombiniert mit entsprechenden Programmier-Praktiken, ist nicht nur ein wichtiger Teil des endgültigen Produktes, sondern kann auch die Entwicklung selbst vereinfachen, schneller und produktiver machen. Der Entwickler sollte die Dokumentation gewissenhaft und gründlich in jeder Stufe der Software-Entwicklung ausführen.

NEU-ENTWICKLUNG

Manchmal hat der Entwickler das letzte an Geschwindigkeit oder das letzte zusätzliche Byte aus einem Programm herauszuholen. In dem Maße, in dem Speicherchips größer wurden, wurde das Speicherproblem weniger ernst. Das zeitliche Problem ist natürlich nur dann von Bedeutung, wenn die Anwendung in dieser Hinsicht kritisch ist. In den meisten Anwendungen benötigt der Mikroprozessor den meisten Teil seiner Zeit zum Warten auf externe Bausteine, und die Programm-Geschwindigkeit ist kein wesentlicher Faktor.

KOSTEN DER NEU-ENTWICKLUNG

Das endgültige Ausfeilen der Eigenschaften eines Programms ist nicht so wesentlich, wie einige Programmierer manchmal glauben. An erster Stelle wird die Neu-Entwicklung aus folgenden Gründen aufwendig:

1) Es wird zusätzliche Programmierzeit benötigt, die häufig die einzigen hohen Kosten in der Software-Entwicklung sind.
2) Die Neu-Entwicklung lohnt sich, wenn die Struktur und Einfachheit des Programms erhöht werden, wodurch sich eine Verringerung der Fehlersuch-Zeit und Testzeit ergibt.
3) Das Programm benötigt zusätzliche Dokumentation.
4) Das sich ergebende Programm wird schwierig zu erweitern, zu warten oder neuerlich zu verwenden sein.

An zweiter Stelle können die niedrigeren Kosten und die besseren Eigenschaften vielleicht nicht so wichtig sein. Werden durch die niedrigeren Kosten und besseren Eigenschaften tatsächlich mehr Geräte gekauft? Oder wäre es besser, mehr anwender-orientierte Eigenschaften unterzubringen? **Die einzigen Anwendungen, die diese zusätzlichen Anstrengungen und Zeit rechtfertigen würden, sind Anwendungen mit sehr hohen Stückzahlen, niedrigen Kosten und einfachen technischen Eigenschaften, bei denen die Kosten eines zusätzlichen Speicherchips die Kosten für die zusätzliche Software-Entwicklung weit übersteigen.** Für alle anderen Anwendungen werden Sie finden, daß man ohne Grund einen unverantwortlich hohen Aufwand treibt.

GRÖSSERE ODER KLEINERE NEU-ORGANISATION

Wenn Sie jedoch ein Programm überarbeiten müssen, so können die folgenden Hinweise von Nutzen sein. Bestimmen Sie zuerst, um wieviel Sie das Programm verbessern müssen oder um wieviel weniger Speicher benötigt wird. Wenn die erforderlichen Verbesserungen 25% oder weniger sind, so könnten Sie den entsprechenden Erfolg durch die Neu-Organisation des Programms erreichen. Liegt es über 25% und Sie haben einen grundlegenden Entwicklungsfehler gemacht, werden Sie wahrscheinlich drastische Änderungen in der Hardware oder Software ausführen müssen. Sie werden sich daher zuerst mit der Neu-Organisation und später mit weitgehenden Änderungen befassen.

Beachten Sie insbesonders, daß die Einsparung von Speicher kritisch sein kann, wenn ein Programm in eine begrenzte Größe von ROM oder RAM passen soll, das auf einem einfachen Einchip- oder Zweichip-Mikrocomputer verfügbar ist. Die Hardware-Kosten für kleine Systeme können dadurch beträchtlich reduziert werden, wenn ihre Anforderungen auf die Speichergröße und E/A-Begrenzungen dieses speziellen Einchip- oder Zweichip-Systems eingeschränkt werden können.

REORGANISATION FÜR WENIGER SPEICHER

EINSPAREN VON SPEICHER

Die folgenden Verfahren werden den Speicherbedarf für Assemblersprachen-Programme des 6502 verringern:

1) **Ersetzen Sie im Programmablauf sich wiederholende Codes (In-Line-Codes) durch Unterprogramme.** Vergewissern Sie sich jedoch, daß die Aufruf- und Rückkehr-Befehle nicht wieder den größten Teil des Gewinnes verbrauchen. Beachten Sie, daß dieser Ersatz gewöhnlich in langsameren Programmen resultiert, da für die Übertragung der Steuerung vorwärts und zurück Zeit aufgewendet werden muß.

2) **Plazieren Sie die am meisten verwendeten Daten auf Seite Null und greifen Sie auf diese mit direkten Einwort-Adressen zu.** Sie können vielleicht auch einige wenige E/A-Adressen hierher plazieren.

3) **Verwenden Sie wenn immer möglich den Stapel.** Der Stapelzeiger wird automatisch nach jeder Verwendung auf den neuesten Stand gebracht, so daß keine ausdrücklichen Befehle für diese Funktion erforderlich sind. Vergessen Sie jedoch nicht, daß der Stapel des 6502 niemals länger als eine Seite sein kann.

4) **Beseitigen Sie Sprung-Anweisungen.** Versuchen Sie das Programm stattdessen entsprechend neu zu organisieren.

5) **Benützen Sie den Vorteil von Adressen, die Sie mit 8-Bit-Größen handhaben können.** Diese beinhalten Speicherplätze auf Seite Null und Adressen, die ein Vielfaches von 100_{16} sind. Z.B. könnten Sie versuchen, alle ROM-Tabellen in einen 100_{16}-Byte-Abschnitt des Speichers zu plazieren und alle RAM-Variablen in einen weiteren 100_{16}-Byte-Abschnitt.

6) **Organisieren Sie Daten und Tabellen so, daß Sie sie adressieren können, ohne sich wegen Überträgen den Kopf zerbrechen zu müssen oder ohne irgendwelche tatsächliche Indizierung.** Dadurch werden Sie wieder 16-Bit-Adressen als 8-Bit-Größen handhaben können.

7) **Verwenden Sie die Bit-Test-Befehle zur Bearbeitung einer Bit-Position an jedem Ende eines Wortes.**

8) **Verwenden Sie übrig gebliebene Ergebnisse von früheren Programm-Abschnitten.**

9) **Benützen Sie den Vorteil von Befehlen wie ASL, DEC, INC, LSR, ROL und ROR, die direkt die Speicherplätze ohne Verwendung von Registern bearbeiten.**

10) **Verwenden Sie INC oder DEC zum Setzen oder Löschen von Flags.**

11) **Verwenden sie relative Sprünge anstatt Sprünge mit direkter Adressierung.**

12) **Achten Sie auf spezielle Kurzformen der Befehle** wie Verschiebungen des Akkumulators (ASL A, LSR A, ROL A und ROR A) und BIT.

13) **Verwenden Sie Algorithmen anstelle von Tabellen** zur Berechnung von arithmetischen oder logischen Ausdrücken und zur Ausführung von Code-Umwandlungen. Beachten Sie, daß dieser Austausch langsamere Programme ergeben kann.

14) **Verringern sie die Größe der mathematischen Tabellen durch Interpolation zwischen den Eingaben.** Hier sparen wir wieder Speicher auf Kosten der Ausführungszeit.

15) **Benützen Sie den Vorteil der CPX- und CPY-Befehle zur Ausführung von Vergleichen ohne den Akkumulator.**

EINSPARUNG VON AUSFÜHRUNGSZEIT

Obwohl einige der Verfahren zur Verringerung des Speicher-Bedarfs auch Zeit einsparen, kann man im allgemeinen nur dann entsprechend viel Zeit einsparen, indem man sich auf häufig ausgeführte Schleifen konzentriert. Auch das vollständige Eliminieren eines Befehls, der nur einmal ausgeführt wird, kann im besten Fall einige wenige Mikrosekunden einsparen. Die Ersparnisse in einer Schleife, die wiederholt ausgeführt wird, gibt ein Mehrfaches hiervon.

Wenn Sie daher die Ausführungszeit verringern müssen, gehen Sie folgendermaßen vor:

1) **Bestimmen Sie, wie häufig jede Programmschleife durchlaufen wird.** Sie können dies von Hand ausführen oder durch Verwendung des Software-Simulators oder anderer Testmethoden.

2) **Prüfen sie die Schleifen in der Reihenfolge ihrer Häufigkeit der Ausführung,** beginnend mit der am meisten verwendeten. Gehen Sie weiter durch diese Liste, bis Sie die erforderliche Verringerung erzielt haben.

3) **Sehen sie zuerst nach, ob es einige Operationen gibt, die außerhalb der Schleife ausgeführt werden,** d.h. wiederholte Berechnungen, Daten, die in ein Register oder in den Stapel plaziert werden können, Daten oder Adressen, die in der Null-Seite gespeichert werden können, spezielle Fälle oder Fehler, die extern verarbeitet werden können, etc. Beachten Sie, daß dies zusätzliche Initialisierung und Speicher erfordert, jedoch eine beträchtliche Einsparung an Zeit bewirkt.

4) **Versuchen Sie Sprung-Adressen zu eliminieren.** Diese benötigen sehr viel Zeit.

5) **Ersetzen Sie Unterprogramme durch einen "In-Line"-Code.** Dies wird wenigstens einen Sprung zu einem Unterprogramm und eine Rückkehr von einen Unterprogramm sparen.

6) **Verwenden Sie den Stapel für zeitweilige Datenspeicherung.**

7) **Verwenden Sie jeden der Hinweise für die Einsparung von Speicher, der auch die Ausführungszeit verringern wird.** Diese beinhalten die Verwendung von 8-Bit-Adressen, BRK, RTI, spezielle Kurzformen von Befehlen etc.

8) **Sehen Sie sich nicht Befehle an, die nur einmal ausgeführt werden.** Jede Änderung, die Sie bei solchen Befehlen machen, bringt meist nur Fehler, und der Gewinn ist ohne Bedeutung.

9) **Vermeiden Sie indizierte und indirekte Adressierung, wann immer möglich,** da diese zusätzliche Zeit benötigen.

10) **Verwenden Sie Tabellen anstelle von Algorithmen.** Bauen Sie die Tabellen so auf, daß Sie soviel wie möglich Aufgaben ausführen können, auch wenn zahlreiche Eingaben wiederholt werden müssen.

GRÖSSERE REORGANISATIONEN

Wenn Sie mehr als 25% Steigerung der Geschwindigkeit oder Verringerung des Speicherbedarfs benötigen, versuchen Sie nicht den Code neu zu organisieren. Ihre Chancen, mehr als einige kleine Verbesserungen zu erzielen, sind gering, außer Sie können einen zusätzlichen Experten hinzuziehen. Sie sind im allgemeinen besser daran, wenn Sie eine größere Änderung durchführen.

BESSERE ALGORITHMEN

Die offensichtlichste Änderung ist ein besserer Algorithmus. Speziell, wenn Sie Dinge wie Sortieren, Suchen oder mathematische Berechnungen ausführen wollen, werden Sie imstande sein, eine schnellere oder kürzere Methode als in der Literatur beschrieben zu finden. Bibliotheken von Algorithmen sind in einigen Zeitschriften und von professionellen Gruppen verfügbar. Sehen Sie sich beispielsweise die Literatur am Ende dieses dieses Kapitels an.

Mehr Hardware kann einen Teil der Software ersetzen. Zähler, Schieberegister, arithmetische Einheiten, Hardware-Multiplikatoren und andere schnelle Zusatzteile können sowohl Zeit wie Speicher sparen. Rechner, UARTs, Tastaturen, Codierer und andere langsame Zusatzgeräte können zwar Speicher sparen, arbeiten dafür aber langsamer. Kompatible parallele und serielle Interface-Schaltungen und andere Bausteine, die speziell zur Verwendung mit dem 6502 oder 6800 geschaffen wurden, können Zeit sparen, indem sie einige Aufgaben der CPU übernehmen.

ANDERE WESENTICHE ÄNDERUNGEN

Andere Änderungen können ebenfalls helfen:

1) **Eine CPU mit einem längeren Wort wird schneller sein,** wenn die Daten lang genug sind. Eine derartige CPU wird insgesamt weniger Speicher benötigen. 16-Bit-Prozessoren verwenden Speicher effizienter als 8-Bit-Prozessoren, da zahlreiche ihrer Befehle ein Wort lang sind.

2) **Es können Versionen der CPU existieren, die mit höheren Taktraten arbeiten.** Aber erinnern Sie sich daran, daß Sie dann ebenfalls schnellere Speicher und E/A-Ports benötigen und daß Sie jede Verzögerungsschleife neu anordnen müssen.

3) **Zwei CPUs können imstande sein, die Aufgabe parallel oder getrennt auszuführen,** wenn Sie die Aufgabe unterteilen können und die Kommunikationsprobleme lösen.

4) **Ein spezieller mikroprogrammierter Prozessor kann imstande sein, das gleiche Programm wesentlich schneller auszuführen.** Die Kosten sind jedoch wesentlich höher, auch wenn Sie eine handelsübliche Emulation benützen.

5) **Sie können Überlegungen zwischen Zeit und Speicher anstellen.** Nachschlage-Tabellen und Funktions-ROMs wären schneller als Algorithmen, würden jedoch mehr Speicher belegen.

Eine derartige Aufgabe, bei der große Verbesserungen erforderlich sind, resultiert gewöhnlich aus mangelnder Planung bei der Definition und in den Entwicklungsstufen. Bei der Stufe der Aufgaben-Definition sollten Sie bestimmen, welcher Prozessor und welche Methoden ausreichend für die Lösung der Aufgabe sind. Wenn Sie eine Fehl-Entscheidung treffen, werden die Kosten später sehr hoch sein. Eine billige Lösung kann in einem unerwarteten Aufwand von kostspieliger Entwicklungszeit resultieren. Hieran werden sie aber nicht vorbeikommen. Die beste Lösung für die richtige Entwicklung und die Vermeidung von Fehlern besteht gewöhnlich in der entsprechenden Erfahrung. Wenn Sie derartige Methoden wie die Aufstellung von Flußdiagrammen, modularer Programmierung, strukturierter Programmierung, Entwicklung mit schrittweiser Verfeinerung und die entsprechende Dokumentation ausgeführt haben, werden Sie imstande sein, eine Menge der aufgewendeten Mühe zu retten, falls Sie eine größere Änderung auszuführen haben.

ENTSCHEIDUNG FÜR EINE GRÖSSERE ÄNDERUNG

LITERATUR

1. Collected Algorithms from ACM. ACM, inc., P. O. Box 12105, Church Street Station, New York 10249.
2. T. C. Chen, "Automatic Computation of Exponentials, Logarithms, , Rations and Square Roots" IBM Journal of Research and Development, Volume 18, pp. 380-388, July, 1972.
3. H. Schmid, Decimal Computation, Wiley-Interscience, New York, 1974.
4. D. E. Knuth, The Art of Computer Programming, Volume 1: Fundamental Algorithms, Addison-Wesley, Reading, Mass., 1967.
5. D. E. Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Addision-Wesley, Reading, Mass., 1969.
6. D. E. Knuth, The Art of Computer Programming, Volume 3: Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.
7. B. Carnahan et al., Applied Numerical Methods, Wiley, New York, 1969.
8. A. M. Despain, "Fourier Transform Computers Using CORDIC Iterations," IEEE Transactions on Computers, Oktober 1974, pp. 993-1001.
9. Y. L. Luke, Algorithms for the Computation of Mathematical Functions, Academic Press, New York, 1977.
10. K. Hwang, Computer-Arithmetic, Wiley, New York, 1978.
11. New methods for performing arithmetic operations on computers are often discussed in the triennial Symposium on Computer Arithmetic. The Proceedings (starting with 1969) are avialable from the IEEE Computer Society, 5855 Naples Plaza, Long Beach, CA 90803.
12. A. D. Edgar, and S. C. Lee "FOCUS Microcomputer Number Systems," Communications of the ACM, March 1979, pp. 166-177.

Kapitel 16
BEISPIELE

Projekt Nr. 1: Eine digitale Stoppuhr

STOPPUHR-EINGABE-VERFAHREN

Zweck: Dieses Projekt stellt eine digitale Stoppuhr dar. Der Bedienende gibt zwei Ziffern (Minuten und Zehntel-Minuten) über eine rechner-ähnliche Tastatur ein und drückt dann die GO-Taste.
Das System zählt die verbleibende Zeit auf zwei Sieben-Segment-LED-Anzeigen abwärts (siehe Kapitel 11 für eine Beschreibung von nicht codierten Tastaturen und LED-Anzeigen).

Hardware: Das Projekt verwendet einen Eingangs- und einen Ausgangsport (einen universellen Interface-Adapter 6522 VIA), zwei Sieben-Segment-Anzeigen, eine Tastatur mit 12 Tasten, einen Inverter 7404 und entweder ein NAND-Gatter 7400 oder ein UND-Gatter 7408, abhängig von der Polarität der Sieben-Segment-Anzeige. Die Anzeigen können Treiber, Inverter und Widerstände erfordern, abhängig von ihrer Polarität und Konfiguration.

Die Hardware ist wie in Bild 16-1 gezeigt konfiguriert. Die Ausgangsleitungen 0, 1 und 2 werden zur Abtastung der Tastatur verwendet. Die Eingangsleitungen 0, 1, 2 und 3 werden verwendet, um zu bestimmen, ob irgendeine der Tasten gedrückt wurde. Die Ausgangsleitungen 0, 1, 2 und 3 dienen zum Senden von BCD-Ziffern zu den Sieben-Segment-Decoder/Treibern. Die Ausgangsleitung 4 wird zum Aktivieren der LED-Anzeigen verwendet (wenn die Leitung 4 gleich "1" ist, so leuchtet die Anzeige). Die Ausgangsleitung 5 dient zur Auswahl der linken oder rechten Anzeige. Die Ausgangsleitung 5 ist "1", wenn die linke Anzeige verwendet wird, "0", wenn die rechte Anzeige verwendet wird. Daher sollte die gemeinsame Leitung an der linken Anzeige aktiv sein, wenn die Leitung 4 gleich "1" und die Leitung 5 gleich "1" ist, während die gemeinsame Leitung an der rechten Anzeige aktiv sein sollte, wenn Leitung 4 gleich "1" und Leitung 5 gleich "0" ist. Die Ausgangsleitung 6 steuert den Dezimalpunkt an der linken Anzeige. Er kann mittels eines Inverters gesteuert werden oder einfach eingeschaltet bleiben.

Tastatur-Verbindungen: Die Tastatur ist eine einfache Rechner-Tastatur, die äußerst billig im Handel erhältlich ist. Sie besteht aus 12 nicht-codierten Tastenschaltern, die in vier Reihen zu je drei Spalten angeordnet sind. Da die Verdrahtung der Tastatur nicht mit den vorgesehenen Reihen und Spalten übereinstimmt, verwendet das Programm eine Tabelle zur Identifizierung der Tasten. Die Tabellen 16-1 und 16-2 enthalten die Eingangs- und Ausgangs-Verbindungen für die Tastatur. Die Taste für den Dezimalpunkt dient zur Bequemlichkeit des Bedienenden und für zukünftige Erweiterungen. Das momentane Programm verwendet diese Taste nicht.

Bei einer tatsächlichen Anwendung würde die Tastatur Pullup-Widerstände benötigen um zu sichern, daß die Eingaben wirklich als logische Einsen gelesen werden, wenn die Tasten nicht gedrückt sind. Es wären ebenfalls strombegrenzende Widerstände oder Treiber mit offenen Kollektoren am Ausgangs-Port erforderlich, um die Beschädigung der VIA-Treiber in dem Fall zu vermeiden, bei dem zwei Ausgänge gegeneinander arbeiten. Dies könnte auftreten, wenn zwei Tasten in der gleichen Reihe zur selben Zeit gedrückt werden, wodurch zwei verschiedene Spalten-Ausgänge miteinander verbunden werden.

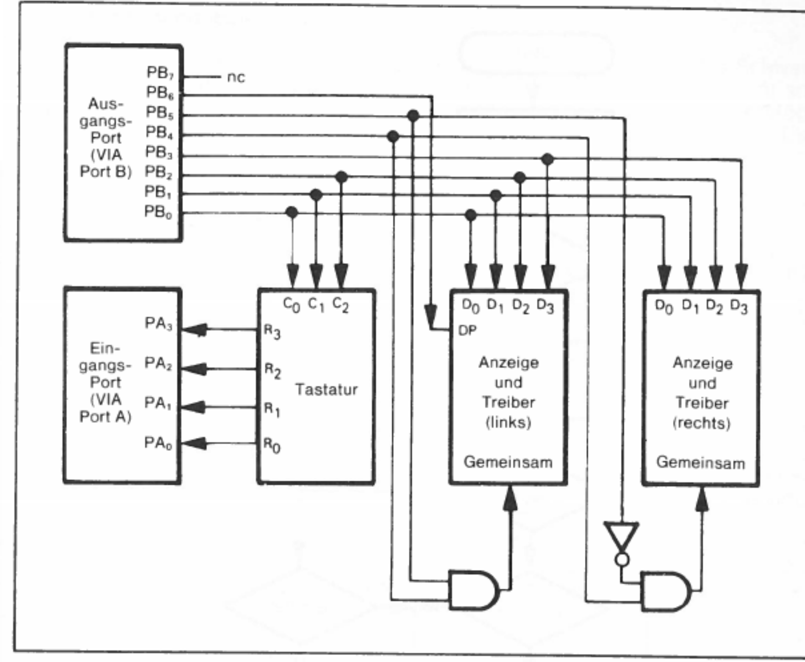


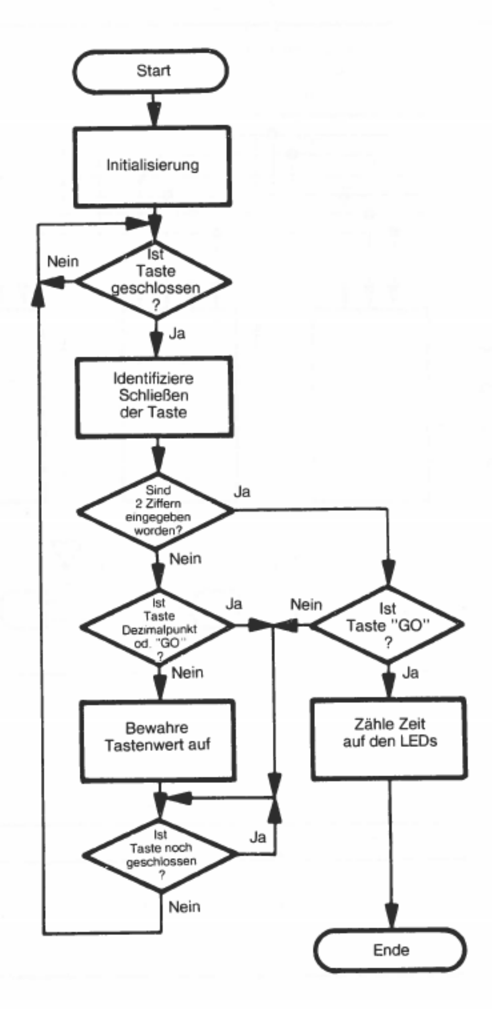
Bild 16-1. E/A-Konfiguration für digitale Stoppuhr.

Tabelle 16-1. Eingangs-Verbindungen für Stoppuhr-Tastatur

Eingangs-Bit	Angeschlossene Tasten
0	'3', '5', '8'
1	'2', '6', '9'
2	'0', '1', '7'
3	'4', '.', 'GO'

Tabelle 16-2. Ausgangs-Verbindungen für Stoppuhr-Tastatur

Ausgangs-Bit	Angeschlossene Tasten
0	'0', '2', '3', '4'
1	'1', '8', '9', 'GO'
2	'5', '6', '7', '.'



Verbindungen für die Anzeige: Die Anzeigen sind standardisierte 7-Segment-Anzeigen mit eingebauten Decodern. Natürlich wären nicht-decodierte 7-Segment-Anzeigen billiger, sie würden jedoch zusätzliche Software benötigen (eine Routine für eine 7-Segment-Umwandlung ist in Kapitel 7 gezeigt). Die Daten werden in die Anzeige als eine einzelne binär codierte Dezimalziffer eingegeben. Die Ziffern werden wie in Bild 11-22 gezeigt dargestellt. Der Dezimalpunkt ist eine einzelne LED, die eingeschaltet wird, wenn der Dezimalpunkt-Eingang logisch 1 ist. Sie können weitere Informationen über Anzeigen in der Literatur 12 und 13 am Ende dieses Kapitels finden.

Programmbeschreibung:

Das Programm ist modular und besitzt mehrere Unterprogramme. Das Schwergewicht wurde mehr auf die Deutlichkeit und die Allgemeingültigkeit gelegt anstatt auf die Effizienz. Offensichtlich nützt das Programm nicht die vollen Möglichkeiten des Prozessors 6502. Jeder Abschnitt der Auflistung wird nun im Detail beschrieben.

1) **Einführende Kommentare**

Die einführenden Kommentare beschreiben das Programm vollständig. Diese Kommentare dienen als Referenz, so daß andere Anwender das Programm leicht anwenden, erweitern und verstehen können. Standardformate, Einrückungen und Abstände erhöhen die Lesbarkeit des Programms.

2) **Definition der Variablen**

Alle Definitionen der Variablen sind an den Anfang des Programms plaziert, so daß sie leicht geprüft und geändert werden können. Jede Variable liegt in einer alphabetischen Liste zusammen mit anderen Variablen der gleichen Art. Kommentare beschreiben die Bedeutung jeder Variablen. Die Kategorien sind:

a) Speichersystem-Konstanten, die von System zu System variieren können, abhängig vom Speicherraum, der unterschiedlichen Programmen oder verschiedenen Speicher-Typen zugewiesen ist.
b) Zeitweilige Speicherung (RAM), die für Variable verwendet wird.
c) E/A- (VIA-) Adressen.
d) Definitionen.

Die Speichersystem-Konstanten sind in den Definitionen enthalten, so daß der Anwender das Programm, zeitweilige Speicher und den Speicherstapel umstellen kann, ohne irgendwelche andere Änderungen ausführen zu müssen. Die Speicher-Konstanten können zur Anpassung an andere Programme geändert werden oder um mit einer speziellen System-Zuweisung von RAM- oder ROM-Adressen übereinzustimmen.

Zeitweilige Speicherung wird mittels Weiterstellung des Stellenzählers (Reserve Memory Byte) zugewiesen, wie in Kapitel 3 gezeigt wurde. Eine = (Equate) Pseudo-Operation plaziert die zeitweiligen Speicherplätze. Eine ORG-Pseudo-Operation plaziert die zeitweiligen Speicherplätze in einen speziellen Teil des Speichers. In diese Speicherplätze werden keine Werte plaziert, so daß das Programm eventuell in ROM oder PROM liegen könnte und das System ohne neuerliches Laden nach einem Einschalten der Betriebspannung arbeiten könnte.

Jede Speicheradresse, die von einem VIA belegt wird, ist so bezeichnet, daß die Adressen leicht für die Handhabung verschiedener Konfigurationen geändert werden können. Die Bezeichnungen dienen auch zur deutlichen Unterscheidung der Steuerregister von Datenregistern.

Die Definitionen machen die Bedeutung bestimmter Konstanten deutlich und gestatten eine einfache Änderung von Parametern. Jede Definition ist in der Form angegeben (binär, hexadezimal, oktal, ASCII oder dezimal), in der ihre Bedeutung am deutlichsten ist. Parameter (wie etwa Entprellzeit) werden hier plaziert, so daß sie entsprechend der Anforderungen des Systems variiert werden können.

3) **Initialisierung**

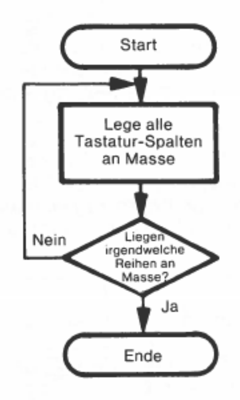
Die Speicherplätze FFFC und FFFD (die RESET-Speicherplätze des 6502) enthalten die Adresse des Startpunktes des Hauptprogrammes. Das Hauptprogramm kann daher überall in den Speicher plaziert und über das RESET-Signal erreicht werden.

Die Initialisierung besteht aus drei Schritten:

a) Plazieren eines Startwertes in den Stapelzeiger. Der Stapel wird nur zur Speicherung der Unterprogramm-Rückkehr-Adressen verwendet. Beachten Sie, daß der Stapelzeiger nur 8 Bit lang ist, da sich der Stapel des 6502 immer auf Seite eins des Speichers befindet.
b) Konfigurieren des VIA.
c) Starten der Anzahl der Zifferntasten, die bei null gedrückt werden.

4) **Ausschauhalten nach einer geschlossenen Taste**

Flußdiagramm:



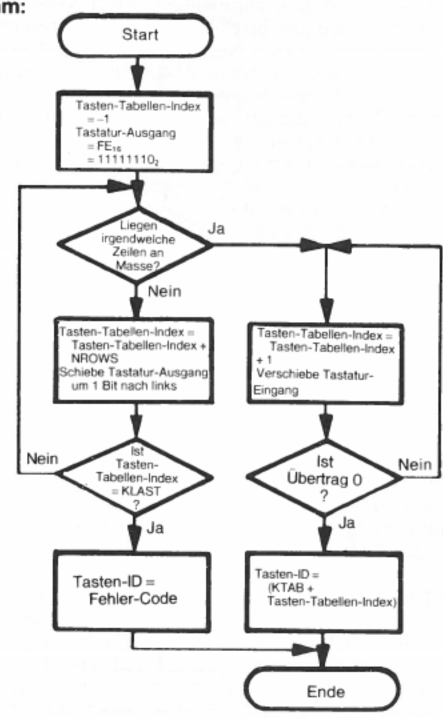
Tastenbetätigungen werden dadurch identifziert, indem alle Tastatur-Spalten auf Masse gelegt werden und dann auf an Masse liegende Reihen geprüft wird (d.h. Spalten-zu-Reihen-Schalter geschlossen). Beachten Sie, daß das Programm nicht annimmt, daß die unbenutzten Eingangs-Bits alle High sind. Stattdessen werden die zu der Tastatur gehörenden Bits mit einem logischen UND-Befehl maskiert.

5) **Tasten-Entprellung**

Das Programm entprellt das Schließen einer Taste mittels Software, indem es zwei Millisekunden wartet. Dies ist im allgemeinen lang genug, um einen einwandfreien Kontakt herzustellen. Das Unterprogramm DELAY zählt einfach mit dem Indexregister X eine Millisekunde lang. Die Anzahl der Millisekunden liegt im Indexregister Y. DELAY müßte neu eingestellt werden, wenn ein langsamerer Takt oder langsamerer Speicher verwendet würden. Sie können diese Änderungen einfach dadurch ausführen, indem Sie die Konstante MSCNT neu definieren.

6) **Identifizieren eines Tasten-Schließens**

Flußdiagramm:



Eine geschlossene Taste wird durch an Masse Legen einzelner Spalten und Beobachten, ob ein Schließen festgestellt wird, identifziert. Sobald ein Schließen festgestellt wurde (so daß die Tastenspalte bekannt ist), kann die Tastenreihe durch Verschieben des Einganges bestimmt werden, bis ein an Masse liegendes Bit gefunden wird.

Das Ausgangsmuster, das erforderlich ist, um einzelne Tastatur-Spalten an Masse zu legen, wird durch Verschieben des ursprünglichen Musters um ein Bit nach links erhalten, nachdem jede Spalte geprüft wurde. Die höchste numerierte Taste der Tastatur wird als eine Marke verwendet, um anzuzeigen, daß alle Spalten an Masse gelegt wurden, ohne daß ein Schließen einer Taste gefunden wurde. Wenn dieser Wert erreicht ist, wird der Fehlercode FF in den Akkumulator plaziert, um anzuzeigen, daß das Hauptprogramm das Schließen nicht identifizieren konnte (d.h. daß das Tasten-Schließen beendet ist oder ein Hardware-Fehler auftrat).

TASTEN-TABELLE

Die Tasten-Identifizierungen liegen in der Tabelle KTAB im Speicher. Die Tasten in der ersten Spalte (die den niedrigstwertigen Ausgangsbits zugewiesen sind) werden von jenen in der zweiten Spalte gefolgt, etc. Innerhalb einer Spalte ist die Taste in der Reihe, die zum niedrigstwertigen Eingangsbit gehört, die erste etc. Daher muß jedesmal beim Abtasten einer Spalte ohne Finden einer geschlossenen Taste die Anzahl der Tasten in einer Spalte (NROWS) zum Tastentabellen-Zeiger addiert werden, um zur nächsten Spalte zu gelangen. Der Tastentabellen-Zeiger wird ebenfalls um eins inkrementiert, bevor jedes Bit in den Reihen-Eingängen geprüft wird. Dieser Vorgang stoppt, wenn ein Null-Eingang gefunden wurde. Beachten Sie, daß der Tastentabellen-Index bei −1 initialisiert wird, da er immer um eins beim Suchen nach der richtigen Reihe inkrementiert wird.

Wenn wir das Schließen einer Taste nicht identifizieren können, ignorieren wir sie einfach und halten nach einer weiteren geschlossenen Taste Ausschau.

7) **Vorgang bei der Tasten-Identifizierung**

Wenn das Programm genügend Ziffern (zwei in diesem einfachen Fall) hat, hält es nur nach der GO-Taste Ausschau und ignoriert alle anderen Tasten. Wenn es eine Zifferntaste findet, bewahrt es den Wert in der Tasten-Anordnung auf, inkrementiert die Anzahl der gedrückten Ziffern-Tasten und inkrementiert den Tastenanordnungs-Zeiger.
Wenn die Eingabe nicht vollständig ist, muß das Programm auf das Ende des Schließens der Taste warten, damit das System nicht die gleiche Tasten-Betätigung neuerlich liest. Der Anwender muß zwischen den Tasten-Betätigungen warten (d.h. eine Taste loslassen, bevor eine andere gedrückt wird). Beachten Sie, daß das Programm doppelte Tasten-Betätigungen als eine oder die andere Taste identifizieren wird, abhängig davon, welche Betätigung die Identifizierungs-Routine zuerst auffindet. Eine verbesserte Version dieses Programms würde die Ziffern so anzeigen, wie sie eingegeben werden und würde dem Anwender gestatten, eine führende oder nacheilende Null wegzulassen (d.h. das Eintasten von ".", "7", "GO", um die Zählung von sieben Zehntel-Minuten zu erhalten).

8) **Einstellung des Anzeige-Ausgangs**

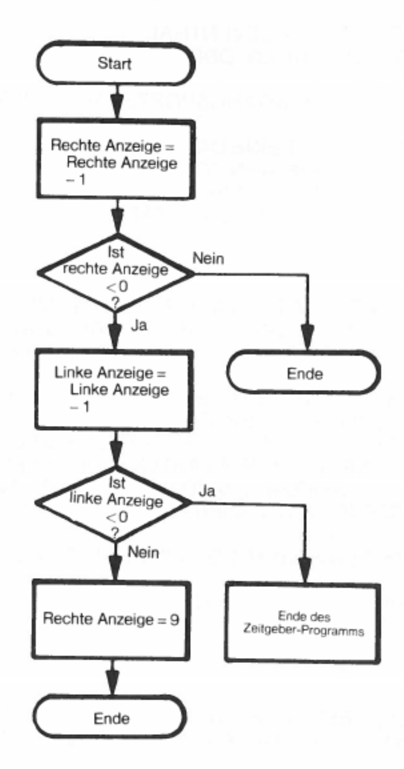
Die Ziffern werden in Register oder Speicherplätze mit gesetztem Bit 4 plaziert, so daß das Ausgangssignal zu den Anzeigen gesendet wird. Die Bits 5 und 6 werden für die höchstwertige Stelle gesetzt, um das Ausgangssignal zur linken Anzeige zu führen und den Dezimalpunkt einzuschalten.

9) **Pulsen der LED-Anzeigen**

Jede Anzeige wird zwei Millisekunden lang eingeschaltet. Dieser Vorgang wird 1500mal wiederholt, um eine gesamte Verzögerung von 0.1 Minuten oder 6 Sekunden zu erhalten. Die Impulse wiederholen sich in so rascher Folge, daß die LED-Anzeigen kontinuierlich eingeschaltet erscheinen.

10) **Dekrementieren der Anzeige-Zählung**

Flußdiagramm:



Der Wert der niedrigstwertigen Stelle wird um eins verringert. Wenn dies Bit 4 beeinflußt (LEDON – verwendet zum Einschalten der Anzeigen), ist die Ziffer negativ geworden. Dann muß ein "Borgen" von der höchstwertigen Stelle erhalten werden. Wenn das Borgen von der höherwertigen Stelle Bit 4 beeinflußt, hat die Zählung null überschritten und der "Countdown" ist beendet. Andernfalls setzt das Programm die niedrigstwertige Stelle auf 9 und läuft weiter.

Beachten Sie, daß die Kommentare beide Abschnitte des Programms und die individuellen Anweisungen beschreiben. Kommentare erklären, was das Programm ausführt, nicht was der spezielle Befehlscode macht. Um die Lesbarkeit zu verbessern, wurden Abstände und Einrückungen verwendet.

```
;NAME DES PROGRAMMS: STOPPUHR
;DATUM: 4/28/79
;PROGRAMMIERER: LANCE A. LEVENTHAL
;PROGRAMM-ANFORDERUNGEN: DD(221) BYTES
;BENÖTIGTES RAM: 5 BYTES
;E/A-ANFORDERUNGEN: 1 EINGANGSPORT, 1 AUSGANGSPORT (1 VIA 6522)
;
;DIESES PROGRAMM STELLT EINE DIGITALE STOPPUHR DAR, DIE EINGA-
;  BEN VON EINER RECHNER-ÄHNLICHEN TASTATUR EMPFÄNGT UND
;  DANN EIN COUNTDOWN AUF ZWEI SIEBEN-SEGMENT-LED-ANZEIGEN IN
;  MINUTEN UND ZEHNTELMINUTEN LIEFERT
;
;TASTATUR
;
;ES WIRD EINE TASTATUR MIT 12 TASTEN ANGENOMMEN
;ES GIBT DREI SPALTEN-VERBINDUNGEN ALS AUSGÄNGE VOM
;  PROZESSOR, SO DASS EINE TASTENSPALTE AN MASSE GELEGT
;  WERDEN KANN
;VIER REIHENVERBINDUNGEN SIND EINGÄNGE ZUM PROZESSOR, SO DASS
;  DIE TASTEN IDENTIFIZIERT WERDEN KÖNNEN
;DIE TASTATUR IST ENTPRELLT DURCH WARTEN FÜR DIE DAUER VON 2
;  MILLISEKUNDEN, NACHDEM EIN TASTENSCHLIESSEN ERKANNT WURDE
;EIN NEUES TASTENSCHLIESSEN WIRD DURCH WARTEN AUF DAS ENDE DES
;  ALTEN SCHLIESSENS IDENTIFIZIERT, DA KEIN TAST-IMPUS VERWENDET
;  WIRD
;DIE TASTATURSPALTEN SIND MIT DEN BITS 0 BIS 2 DES PORTS B DES VIA
;  VERBUNDEN
;DIE TASTATURREIHEN SIND MIT DEN BITS 0 BIS 3 DES PORTS A DES VIA
;  VERBUNDEN
;
;ANZEIGEN
;
;ES WERDEN ZWEI SIEBEN-SEGMENT-LED-ANZEIGEN MIT GETRENNTEN
;  DECODERN (7447 ODER 7448, ABHÄNGIG VON DER ART DER ANZEIGEN)
;  VERWENDET
;DIE DATEN-EINGÄNGE DES DECODERS SIND MIT DEN BITS 0 BIS 3 DES
;  B-PORTS DES VIA VERBUNDEN
;BIT 4 DES B-PORTS DES VIA WIRD ZUM AKTIVIEREN DER LED-ANZEIGEN
;  VERWENDET (BIT 4 IST 1 ZUM SENDEN DER DATEN ZU DEN LEDS)
;BIT 5 DES B-PORTS DES VIA WIRD ZUR AUSWAHL VERWENDET,
;  WELCHE LED GERADE VERWENDET WIRD (BIT 5 IST 1, WENN DIE
;  FÜHRENDE ANZEIGE VERWENDET WIRD, 0 WENN DIE FOLGENDE
;  ANZEIGE VERWENDET WIRD)
;BIT 6 DES B-PORTS DES VIA WIRD ZUM EINSCHALTEN DES
;  DEZIMALPUNKTES AUF DER FÜHRENDEN ANZEIGE VERWENDET (BIT 6
;  IST 1, WENN DIE ANZEIGE EINZUSCHALTEN IST)
;
```



```
;VERFAHREN
;
;SCHRITT 1 – INITIALISIERUNG
;  DER SPEICHERSTAPEL-ZEIGER (VERWENDET FÜR UNTERPROGRAMM-
;  RÜCKKEHRADRESSEN) WIRD INITIALISIERT. DIE ANZAHL DER
;  GEDRÜCKTEN ZIFFERNTASTEN WIRD AUF NULL GESETZT.
;SCHRITT 2 – FESTSTELLEN EINES TASTENSCHLIESSENS
;  ALLE TASTATURSPALTEN WERDEN AN MASSE GELEGT, UND DIE
;  TASTATURREIHEN WERDEN GEPRÜFT, BIS EINE GESCHLOSSENE TASTE
;  GEFUNDEN WIRD
;SCHRITT 3 – ENTPRELLEN DER TASTEN
;  EIN WARTEN VON 2 MS WIRD ZUM ELIMINIEREN DES TASTENPRELLENS
;  EINGEFÜHRT
;SCHRITT 4 – IDENTIFIZIERUNG EINES TASTENSCHLIESSENS
;  DAS TASTENSCHLIESSEN WIRD DURCH LEGEN EINER EINZELNEN
;  TASTATURSPALTE AN MASSE UND BESTIMMEN DER REIHE UND SPALTE
;  IDENTIFIZIERT. ES WIRD EINE TABELLE ZUM CODIEREN DER TASTEN
;  ENTSPRECHEND IHRER REIHEN- UND SPALTENNUMMERN VERWENDET
;  IN DER TASTENTABELLE WERDEN DIE ZIFFERN DURCH IHRE WERTE
;  IDENTIFIZIERT. DIE DEZIMALPUNKT-TASTE IST NUMMER 10 UND DIE
;  "GO"-TASTE IST NUMMER 11
;SCHRITT 5 – AUFBEWAHREN DES TASTENSCHLIESSENS
;  DAS SCHLIESSEN DER TASTEN WIRD IN DER ZIFFERNTASTEN-
;  ANORDNUNG AUFBEWAHRT, BIS ZWEI ZIFFERN IDENTIFIZIERT WURDEN.
;  DEZIMALPUNKTE, WEITERE ZIFFERN UND BETÄTIGUNGEN DER
;  "GO"-TASTE WERDEN, BEVOR ZWEI ZIFFERN EINGEGEBEN WURDEN,
;  IGNORIERT.
;  NACHDEM ZWEI ZIFFERN GEFUNDEN WURDEN, WIRD DIE
;  "GO"-TASTE ZUM STARTEN DES COUNTDOWN-VORGANGS VERWENDET
;SCHRITT 6 – COUNTDOWN-ZEITGEBERINTERVALL AUF DEN LEDS
;  EIN COUNTDOWN WIRD AUF DEN LEDS AUSGEFÜHRT, WOBEI DIE
;  FÜHRENDE STELLE DIE VERBLEIBENDE ZAHL DER MINUTEN UND DIE
;  FOLGENDE STELLE DIE ZAHL DER VERBLEIBENDEN ZEHNTELMINUTEN
;  DARSTELLT
;
;
;DEFINITIONEN DER STOPPUHR-VARIABLEN
;
;SPEICHERSYSTEM-KONSTANTEN
;
BEGIN   =$0400                  ;BEGINN IST DER START-SPEICHERPLATZ
                                ;  FÜR DAS PROGRAMM
STKBGN  =$FF                    ;BEGINN DER STAPELADRESSE AUF SEITE 1
TEMP    =0                      ;STARTADRESSE FÜR DEN RAM-SPEICHER
;
;ZEITWEILIGER RAM-SPEICHER
;
        *=TEMP
DCTR    *=*+2                   ;16-BIT-ZÄHLER FÜR ZEITSCHLEIFE
KEYNO   *=*+2                   ;ZIFFERNTASTEN-ANORDNUNG – SIE
                                ;  BEWAHRT DIE IDENTIFIZIERUNGEN DER
                                ;  ZIFFERNTASTEN AUF, DIE GEDRÜCKT
                                ;  WORDEN SIND
NKEYS   *=*+1                   ;ANZAHL DER GEDRÜCKTEN
                                ;  ZIFFERNTASTEN
;
```

```
;E/A-EINHEITEN UND VIA-ADRESSEN
;
VIAORB   =$A000             ;AUSGANGS-PORT FÜR TASTATUR UND
                            ;  ANZEIGE
VIAORA   =$A001             ;EINGANGS-PORT FÜR TASTATUR
VIADDRB  =$A002             ;DATEN-RICHTUNGS-REGISTER FÜR PORT B
VIADDRA  =$A003             ;DATEN-RICHTUNGS-REGISTER FÜR
                            ;  PORT A
VIAPCR   =$A00C             ;PERIPHERESSTEUERREGISTER DES VIA
;
;DEFINITIONEN
;
DECPT    =%01000000         ;CODE ZUM EINSCHALTEN DER
                            ;  DEZIMALPUNKT-LED
ECODE    =$FF               ;FEHLERCODE, WENN ID-ROUTINE DIE
                            ;  TASTE NICHT FINDET
GOKEY    =11                ;IDENTIFIZIERUNGS-NUMMER FÜR
                            ;  "GO"-TASTE
KLAST    =11                ;HÖCHSTE NUMERIERTE TASTE
LEDON    =%00010000         ;CODE ZUM SENDEN VON AUSGABEN ZU
                            ;  DEN LEDS
LEDSL    =%00100000         ;CODE ZUR AUSWAHL DER FÜHRENDEN
                            ;  ANZEIGE
MSCNT    =$C7               ;ZÄHLUNG, ERFORDERLICH FÜR EINE
                            ;  VERZÖGERUNG VON 1 MS
MXKEY    =2                 ;MAXIMALE ANZAHL DER VERWENDETEN
                            ;  ZIFFERNTASTEN-SCHLIESSUNGEN
NROWS    =4                 ;ANZAHL DER REIHEN IN DER TASTATUR
OPEN     =%00001111         ;EINGABE VON TASTATUR, WENN KEINE
                            ;  TASTE GESCHLOSSEN
TPULS    =2                 ;ANZAHL DER MS ZWISCHEN DEN
                            ;  ZIFFERN-ANZEIGEN
TWAIT    =2                 ;ANZAHL DER MS ZUM ENTPRELLEN DER
                            ;  TASTEN
;
;
;
;
         *=$FFC
;
;RÜCKSETZ-ADRESSE, UM ZUM STOPPUHR-PROGRAMM ZU GELANGEN
;
         .WORD  BEGIN       ;VEKTOR ZUM STARTEN DES
                            ;  STOPPUHR-PROGRAMMS
;
```



```
;INITIALISIERUNG DES STOPPUHR-PROGRAMMS
;
         *=BEGIN
         LDA   #0
         STA   VIADDRA      ;MACHE VIA A-PORT ZU
                            ;  EINGANGSLEITUNGEN
         STA   VIAPCR       ;MACHE ALLE STEUERLEITUNGEN ZU
                            ;  EINGÄNGEN
         LDA   #$FF
         STA   VIADDRB      ;MACHE VIA B-PORT ZU
                            ;  AUSGANGSLEITUNGEN
         LDX   #STKBGN      ;INITIALISIERE STAPELZEIGER
         TXS
;
;INITIALISIERE ZIFFERNTASTEN-ZÄHLER
;RT ZU
;  AUSGANGSLEITUNGEN
         LDX   #STKBGN      ;INITIALISIERE STAPELZEIGER
         TXS
;
;INITIALISIERE ZIFFERNTASTEN-ZÄHLER
;
INITL    LDA   #0           ;ANZAHL DER ZIFFERNTASTEN = NULL
         STA   NKEYS
;
;ABTASTEN DER TASTATUR UND AUSSCHAU HALTEN NACH
;  GESCHLOSSENEN TASTEN
;
SRCHK    JSR   SCANC        ;WARTE AUF SCHLIESSEN EINER TASTE
;
;WARTE AUF DAS ENTPRELLEN EINER TASTE
;
         LDY   #TWAIT       ;HOLE ENTPRELLZEIT IN MS
         JSR   DELAY        ;WARTE AUF EINE TASTE ZUM STOPPEN
                            ;  DES ENTPRELLENS
;
;IDENTIFIZIERE DIE GEDRÜCKTE TASTE
;
         JSR   IDKEY        ;IDENTIFIZIERE DIE GESCHLOSSENE TASTE
         CMP   #ECODE       ;WELCHE TASTE WURDE IDENTIFIZIERT?
         BEQ   SRCHK        ;NEIN, WARTE AUF DAS NÄCHSTE
                            ;  SCHLIESSEN
;
;STARTE ZEITZÄHLUNG, WENN "GO"-TASTE GEDRÜCKT UND GENÜGEND
;  ZIFFERN VORLIEGEN
;
         LDX   NKEYS        ;WURDE DIE MAXIMALE ANZAHL VON
                            ;  ZIFFERNTASTEN EINGEGEBEN?
         CPX   #MXKEY
         BNE   ENTDG        ;NEIN, GEHE ZUR EINGABE DER
                            ;  ZIFFERNTASTE
         CMP   #GOKEY       ;JA, IST TASTE "GO"?
         BEQ   STTIM        ;JA, STARTE ZEIT-ZÄHLUNG
         BNE   WAITK        ;NEIN, IGNORIERE TASTE
;
```

```
;GIB ZIFFERNTASTE IN DIE ANORDNUNG EIN, IGNORIERE DEZIMALPUNKT
;  ODER "GO"-TASTE
;
ENTDG   CMP   #10        ;IST TASTE EINE ZIFFER?
        BCS   WAITK      ;NEIN, IGNORIERE SIE
        INC   NKEYS      ;JA, INKREMENTIERE ANZAHL DER
                         ;  EINGEGEBENEN ZIFFERNTASTEN
        STA   KEYNO,X    ;BEWAHRE ZIFFERNTASTE IN ANORDNUNG
                         ;  AUF
;
;WARTE AUF ENDE DES MOMENTANEN TASTEN-SCHLIESSENS
;
WAITK   JSR   SCANO      ;WARTE BIS ALLE TASTEN LOSGELASSEN
                         ;  SIND
        BEQ   SRCHK      ;HALTE NACH NÄCHSTEM
                         ;  TASTEN-SCHLIESSEN AUSSCHAU
                         ;  (SCANO SETZT IMMER Z)
;
;VERARBEITEN DER ZIFFERN FÜR DIE ANZEIGE
;
STTIM   LDA   KEYNO      ;HOLE FÜHRENDE ZIFFERN
        ORA   #DECPT     ;SCHALTE DEZIMALPUNKT FÜR FÜHRENDE
                         ;  ZIFFER EIN
        ORA   #LEDON     ;SETZE AUSGANGS-SIGNAL ZU DEN LEDS
        ORA   #LEDSL     ;WÄHLE FÜHRENDE ANZEIGE
        STA   KEYNO
        LDA   KEYNO+1    ;HOLE FOLGENDE ZIFFER
        ORA   #LEDON     ;SETZE AUSGANGSSIGNAL FÜR LEDS
        STA   KEYNO+1
;
;PULSEN DER LED-ANZEIGE
;
LEDLP   LDA   #6         ;SETZE ZÄHLER FÜR 1500 IMPULSE
        STA   DCTR+1
TLOOP   LDA   #250
        STA   DCTR
DSPLY   LDA   KEYNO      ;SENDE FÜHRENDE ZIFFER ZU LED 1
        STA   VIAORB
        LDY   #TPULS     ;VERZÖGERUNG ZWISCHEN ZIFFERN
        JSR   DELAY
        LDA   KEYNO+1    ;SENDE FOLGENDE ZIFFER ZUR LED 2
        STA   VIAORB
        LDY   #TPULS     ;VERZÖGERUNG ZWISCHEN ZIFFERN
        JSR   DELAY
        DEC   DCTR
        BNE   DSPLY
        DEC   DCTR+1
        BNE   TLOOP
;
```



```
;DEKREMENTIERE ZÄHLUNG AUF DEN LED-ANZEIGEN
;
        LDA   #LEDON     ;HOLE BIT-MUSTER UND SCHAUE NACH
                         ;  ÜBERTRAG
        DEC   KEYNO+1    ;ZÄHLE FOLGENDE ZIFFER ABWÄRTS
        BIT   KEYNO+1    ;HAT DIE FOLGENDE ZIFFER NULL
                         ;  UNTERSCHRITTEN?
        BNE   LEDLP      ;NEIN, ZEIGE NEUE ZEIT AN
        DEC   KEYNO      ;JA, ZÄHLE FÜHRENDE ZIFFER ABWÄRTS
        BIT   KEYNO      ;IST FÜHRENDE ZIFFER UNTER NULL?
        BEQ   INITL      ;JA, WARTE AUF NÄCHSTE
                         ;  ZEITGEBER-AUFGABE
        LDA   #9         ;NEIN, MACHE FOLGENDE ZIFFER 9
        ORA   #LEDON     ;SETZE AUSGANGSSIGNAL ZU DEN LEDS
        STA   KEYNO+1
        BNE   LEDLP      ;KEHRE ZUM PULSEN DER ANZEIGEN
                         ;  ZURÜCK
;
;DAS UNTERPROGRAMM SCANC TASTET DIE TASTATUR AB UND WARTET
;  AUF DAS SCHLIESSEN EINER TASTE
;  ALLE TASTATUR-EINGÄNGE LIEGEN AN MASSE
;
SCANC   LDA   #0         ;LEGE ALLE TASTATUR-SPALTEN AN
                         ;  MASSE
        STA   VIAORB
KEYCLS  LDA   VIAORA     ;HOLE TASTATURREIHEN-DATEN
        AND   #OPEN      ;IGNORIERE UNBENÜTZTE EINGÄNGE
        CMP   #OPEN      LIEGEN IRGENDWELCHE EINGÄNGE
                         ;  AN MASSE?
        BEQ   KEYCLS     ;NEIN, WARTE
        RTS
;
;DAS UNTERPROGRAMM DELAY WARTET AUF DIE ANZAHL DER MS, DIE IM
;  INDEX-REGISTER Y DURCH ZÄHLUNG MIT DEM INDEX-REGISTER X
;  SPEZIFIZIERT SIND
;
DELAY   LDX   #MSCNT     ;ZÄHLE FÜR EINE 1 MS-VERZÖGERUNG
WTLP    DEX              ;WARTE 1 MS
        BNE   WTLP
        DEY              ;ZÄHLE MS
        BNE   DELAY
        RTS
;
;DAS UNTERPROGRAMM IDKEY BESTIMMT DIE REIHEN- UND
;  SPALTENNUMMER DER GESCHLOSSENEN TASTE UND IDENTIFIZIERT DIE
;  TASTE AUS EINER TABELLE
;
IDKEY   LDA   #%11111110 ;HOLE MUSTER ZUM LEGEN DER SPALTE
                         ;  NULL AN MASSE
        STA   VIAORB     ;LEGE SPALTE NULL AN MASSE
        LDX   #$FF       ;TASTEN-TABELLEN-INDEX = -1
```

```
FCOL    LDA   VIAORA      ;HOLE TASTATURREIHEN-DATEN
        AND   #OPEN       ;IGNORIERE UNBENÜTZTE EINGÄNGE
        CMP   #OPEN       ;LIEGEN IRGENDWELCHE EINGÄNGE AN
                          ;  MASSE?
        BNE   FROW        ;JA, BESTIMME WELCHE REIHE AN MASSE
                          ;  LIEGT
        ROL   VIAORB      ;NEIN, VERSCHIEBE AUSGANG ZUM LEGEN
                          ;  DER NÄCHSTEN SPALTE AN MASSE
        TXA               ;BRINGE TASTENTABELLEN-ZEIGER ZUR
                          ;  NÄCHSTEN SPALTE
        ADC   #NROWS-1
        TAX
        CMP   #KLAST      ;WURDEN ALLE SPALTEN GEPRÜFT?
        BNE   FCOL        ;NEIN, PRÜFE NÄCHSTE SPALTE
        LDA   #ECODE      ;JA, KEHRE MIT FEHLERCODE IN A
                          ;  ZURÜCK
        RTS
;
;
;BESTIMME REIHENNUMMER DER GESCHLOSSENEN TASTE
;
FROW    INX               ;INKREMENTIERE TASTENTABELLEN-INDEX
        LSR   A           ;SCHIEBE EINGÄNGE ZUR BESTIMMUNG
                          ;  DER AN MASSE LIEGENDEN REIHE
        BCS   FROW
;
;IDENTIFIZIERE DIE TASTE AUS DER TABELLE
;
        LDA   KTAB,X      HOLE TASTENNUMMER AUS TABELLE
        RTS
;
;TASTATUR-TABELLE
;
;SPALTEN SIND DER PRIMÄRE INDEX, REIHEN DER SEKUNDÄRE INDEX.
;DIE TASTEN IN DER SPALTE, DIE MIT AUSGANGSBIT 0 VERBUNDEN SIND,
;  WERDEN VON JENEN IN DER SPALTE GEFOLGT, DIE ZUM AUSGANGSBIT
;  1 GEHÖREN, ETC. INNERHALB EINER SPALTE IST DIE MIT DEM
;  EINGANGSBIT 0 VERBUNDENE TASTE DIE ERSTE, GEFOLGT VON DER MIT
;  EINGANGSBIT 1 VERBUNDENEN ETC.
;
;DIE ZIFFERNTASTEN SIND 0 BIS 9. DER DEZIMALPUNKT IST 10 UND "GO"
;  IST 11
;
KTAB    .BYTE             3,2,0,4,8,9,1,11,5,6,7,10
;
;DAS UNTERPROGRAMM SCANO WARTET, BIS ALLE TASTEN LOSGELASSEN
;  SIND, SO DASS DIE NÄCHSTE TASTENBETÄTIGUNG GEFUNDEN WERDEN
;  KANN
;
SCANO   LDA   #0          ;LEGE ALLE TASTATURSPALTEN AN MASSE
        STA   VIAORB
KEYOPN  LDA   VIAORA      ;HOLE TASTATURREIHEN-EINGÄNGE
        AND   #OPEN       ;IGNORIERE UNBENUTZTE EINGÄNGE
        CMP   #OPEN       ;SIND IRGENDWELCHE TASTEN
                          ;  GEDRÜCKT?
        BNE   KEYOPN      ;JA, WARTE BIS ALLE LOSGELASSEN
        RTS
        .END
```


Projekt Nr. 2: Ein Digital-Thermometer

Zweck: Dieses Projekt stellt ein Digital-Thermometer dar, das die Temperatur in Grad Celsius auf zwei Sieben-Segment-Anzeigen darstellt.

Hardware: Das Projekt verwendet einen Eingangs- und einen Ausgangsport, zwei Sieben-Segment-Anzeigen, einen Inverter 74LS04, ein NAND-Gatter 74LS00 oder ein UND-Gatter 74LS08, abhängig von der Polarität der Anzeigen, einen monolitischen A/D-Konverter mit 8 Bits AD7570J von Analog Devices, einen Komparator LM311 und verschiedene periphere Treiber, Widerstände und Kondensatoren, die von den Anzeigen und dem Konverter benötigt werden. (Siehe Kapitel 11 für eine Besprechung von A/D-Wandlern. Siehe auch Literatur 1 am Ende dieses Kapitels für die Besprechung von A/D-Wandlern.)

Bild 16-2 zeigt die Organisation der Hardware. Die Steuerleitung CB2 vom VIA wird zum Senden eines Signals "Start der Umwandlung" zum A/D-Konverter verwendet. Die Eingangsleitungen 0 bis 7 sind direkt mit acht digitalen Datenleitungen vom Konverter verbunden. Die Ausgangsleitungen 0 bis 3 dienen zum Senden von BCD-Ziffern zu den Sieben-Segment-Decodern/Treibern. Die Ausgangsleitung 4 aktiviert die Anzeigen, und die Ausgangsleitung 5 wählt die linke oder rechte Anzeige aus (Leitung 5 ist "1" für die linke Anzeige). Die Steuerleitung CA1 wird dazu verwendet, um zu bestimmen, wann die Umwandlung abgeschlossen ist (BUSY wird eins).

ANALOG-HARDWARE DES THERMOMETERS

Der Analogteil der Hardware ist in Bild 16-3 dargestellt. Der Thermistor bildet einfach einen Widerstand, der von der Temperatur abhängt. Bild 16-4 ist ein Diagramm des Widerstandes, und Bild 16-5 zeigt den Bereich des Stromes, bei dem der Widerstand konstant ist. Die Umwandlung in Grad Celsius im Programm wird mit einer Eichtabelle ausgeführt. Die verschiedenen Potentiometer können zum exakten Eichen der Daten verwendet werden. Ein Taktsignal für den A/D-Wandler wird von einem RC-Netzwerk erzeugt, wie in Bild 16-3 gezeigt ist. Die Werte sind R7 = 33kΩ und C = 1000 pF, so daß die Taktfrequenz etwa bei 75 KHz liegt. Bei dieser Frequenz beträgt die maximale Umwandlungszeit für 8 Bits etwa 50 Mikrosekunden. Wenn BUSY auf High geht, setzt es Bit 1 des Unterbrechungsflag-Registers des VIA. Die 8-Bit-Version des Konverters benötigt folgende spezielle Verbindungen. Die acht Datenleitungen sind DB2 bis DB9 (DB1 ist immer High während der Umwandlung und DB0 Low).

Der 8-Bit-Eingang "Short Cycle" (Anschluß 26-SC8) wird auf Low gezogen, so daß im vorliegenden Fall nur eine 8-Bit-Umwandlung ausgeführt wird. Die Anschlüsse "High Byte Enable" (Anschluß 20-HBEN) und "Low Byte Enable" (Anschluß 21-LBEN) werden beide auf High gezogen, so daß die Daten-Ausgänge immer freigegeben sind.

Der Wandler verwendet das Verfahren der Stufenumsetzung. Es vergleicht jedes Bit mit dem Ausgang eines D/A-Wandlers um festzustellen, ob dieses Bit ein- oder ausgeschaltet sein sollte. Jeder Vergleich benötigt eine Taktperiode. Dieses Verfahren ist sehr schnell, jedoch fehleranfällig, wenn das Eingangssignal verrauscht ist. Genauere Methoden für die Handhabung analoger E/A sind in der Literatur am Ende des Kapitels beschrieben.

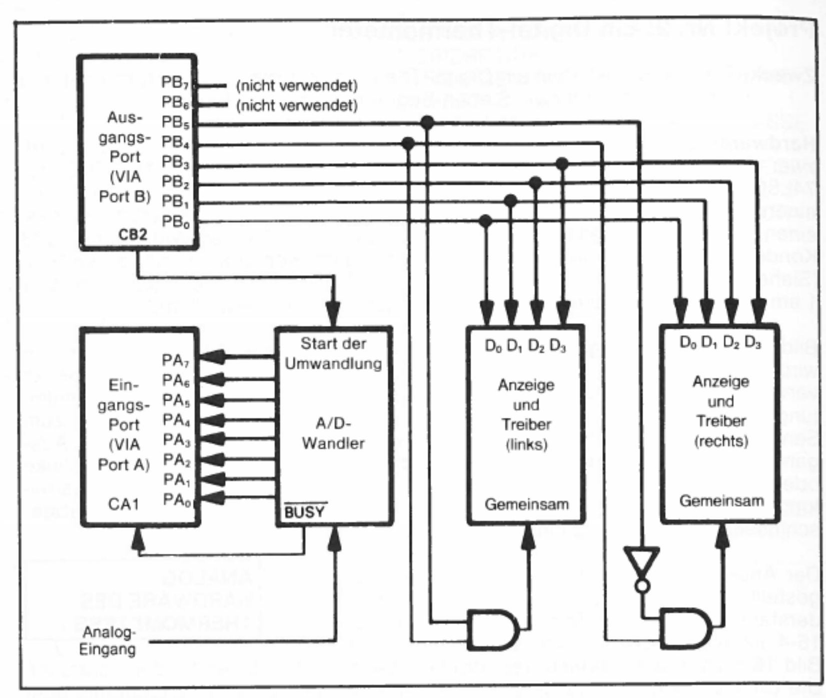


Bild 16-2. E/A-Konfiguration für ein Digital-Thermometer.

Anmerkung: Wenn eine positive V_{REF} verwendet wird, so reicht der Analog-Eingangs-Bereich von 0 bis $-V_{REF}$ und der Komparator-(–)-Eingang sollte mit OUT1 (Anschluß 4) des AD7570 verbunden werden.,

R_T ist der Thermistor. Das Analog-Eingangssignal vom Spannungsteiler beträgt:

$$\frac{R_F}{R_F + R_T} \times 15\ V$$

Da $R_F = 68\ k\Omega$, beträgt die Eingangsspannung

$$\frac{1{,}02 \times 10^6}{R_F + 6{,}8 \times 10^4}\ \text{Volt}$$

Der Mindestwert von R_T beträgt $3{,}4 \times 10^4\ \Omega$ (T = 50C, siehe Bild 16-4) so daß der Gesamtmeßbereich 10 Volt beträgt.

Bild 16-3. Analog-Hardware des Digital-Thermometers.

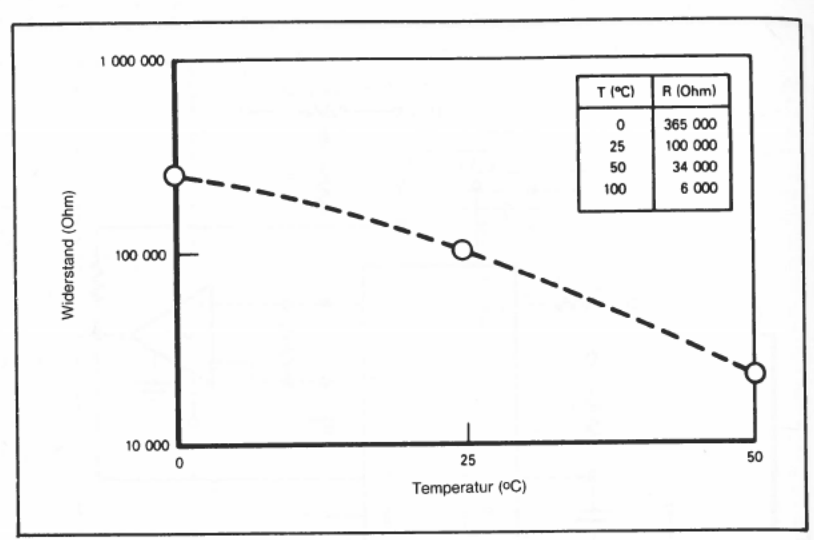


Bild 16-4. Kennlinie des Thermistors (Fenwal GA51J1 Bead).

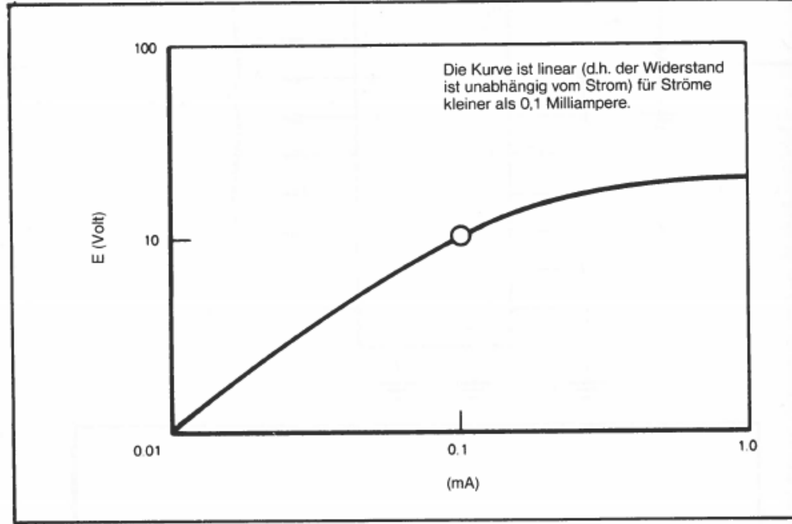
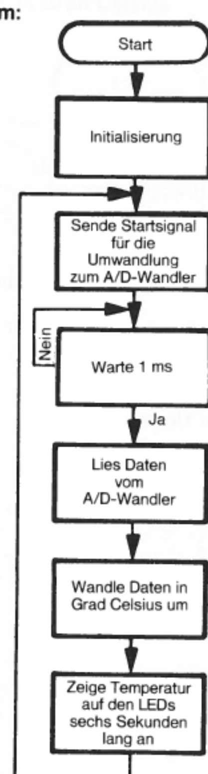


Bild 16-5. Typische E-I-Kennlinie für den Thermistor (25 °C).

Allgemeines Flußdiagramm:



Programm-Beschreibung:

1) **Initialisierung**

Die Speicherplätze FFFC und FFFD (die Reset-Speicherplätze des 6502) enthalten die Start-Adresse des Programms. Die Initialisierung konfiguriert den VIA und plaziert einen Wert in den Stapelzeiger. Der Stapel wird nur zur Speicherung der Rückkehr-Adressen der Unterprogramme verwendet. Erinnern Sie sich daran, daß Kapitel 12 zahlreiche Beispiele zur Konfiguration des VIA enthält.

2) **Senden des Startsignals zum A/D-Wandler für die Umwandlung**

Die CPU liefert das Startsignal für die Umwandlung, indem sie zuerst eine Eins und dann eine Null auf die Ausgangsleitung CB2 legt. Jedes Eingangssignal vom Wandler benötigt einen Start-Impuls.

3) **Warten auf den Abschluß der Umwandlung**

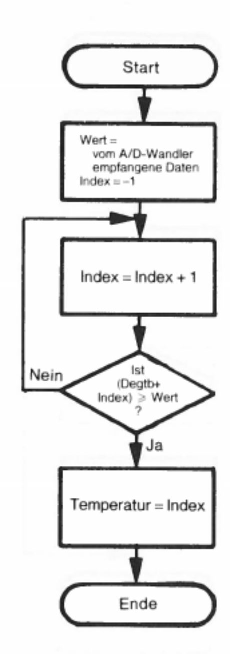
Ein Übergang von "0" auf "1" auf der $\overline{\text{BUSY}}$-Leitung setzt Bit 1 des VIA-Unterbrechungsflag-Registers. In Wirklichkeit benötigt der Wandler nur maximal 50 Mikrosekunden für eine 8-Bit-Umwandlung, so daß eine kurze Verzögerung ebenfalls ausreichend wäre. Beachten Sie, daß das Lesen der Wandler-Daten Bit 1 des VIA-Unterbrechungsflag-Registers löscht, so daß die nächste Operation ordnungsgemäß ablaufen kann.

4) **Lesen der Daten vom A/D-Wandler**

Das Lesen der Daten beinhaltet eine einzelne Eingabe-Operation. Wir sollten beachten, daß der Analog-Baustein AD7570J einen Freigabe-Eingang und Tristate-Ausgänge besitzt, so daß er direkt mit dem Datenbus des Mikroprozessors verbunden werden könnte.
Der Wandler 7570 wird natürlich in dieser speziellen Anwendung nicht voll genützt, insbesonders da wir ihn an einen 6502-Prozessor über ein VIA anschließen. Ein einfacher 8-Bit-Wandler wie der im Kapitel 11 beschriebene 5357 würde die Aufgabe billiger lösen.

5) **Umwandlung der Daten in Grad Celsius**

Flußdiagramm:



VERWENDUNG EINER EICH-TABELLE

Die Umwandlung verwendet eine Tabelle, die den größten Eingangswert entsprechend einer gegebenen Temperatur enthält. Das Programm sucht die Tabelle und hält nach einem Wert Ausschau, der größer oder gleich dem vom Konverter empfangenen Wert ist. Der erste derart aufgefundene Wert entspricht der geforderten Temperatur, das heißt, wenn die zehnte Eingabe der erste Wert ist, der gleich oder größer als die Daten ist, beträgt die Temperatur 10 Grad. Dieses Suchverfahren ist nicht sehr effizient, reicht jedoch für die vorliegende Anwendung vollkommen aus.

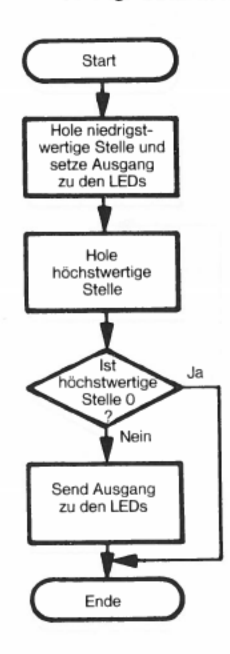
Das Umwandlungs-Unterprogramm gibt einen Binärwert zurück, der dann in BCD durch wiederholtes Subtrahieren von Zehn umgewandelt wird, bis der Rest negativ wird. Die letzte Zehn wird dann zurück addiert, um die niedrigstwertige Stelle zu bilden.

Die Tabelle könnte durch eine Eichung aufgestellt werden oder durch ein mathematisches Näherungsverfahren. Das Eichverfahren ist einfacher, da das Thermometer sonst auf andere Weise geeicht werden müßte. Die Tabelle belegt einen Speicherplatz für jeden Temperaturwert. Ein Verfahren, das wesentlich weniger Speicher verwendet, ist in den Literaturquellen 1 und 2 beschrieben.

Um das Thermometer zu eichen, müssen Sie zuerst die Potentiometer einstellen, damit der Gesamtbereich stimmt und dann die Ausgangswerte des Wandlers für die entsprechende Temperatur bestimmen.

6) **Vorbereitung der Daten für die Anzeige**

Flußdiagramm:



AUSBLENDEN EINER FÜHRENDEN NULL

Für die niedrigstwertige Stelle setzen wir einfach das Bit, das die Anzeige einschaltet. Das Ergebnis wird im RAM-Speicherplatz LSTEMP aufbewahrt.

Der einzige Unterschied für die höchstwertige Stelle besteht darin, daß eine führende Null ausgeblendet wird (das heißt, die Anzeige zeigt "7" anstatt "07" für 7°C). Dies beinhaltet einfach, daß das Bit nicht gesetzt wird, welches die Anzeige einschaltet, wenn die Stelle null ist. Das Ergebnis wird in der Nullseiten-Adresse MSTEMP aufbewahrt.

7) **Anzeige der Temperatur während der Dauer von sechs Sekunden**

Flußdiagramm:



Jede Anzeige wird häufig genug gepulst, so daß sie scheinbar ständig leuchtet. Wenn TPULS größer gemacht wird (beispielsweise 50 ms), würde die Anzeige flackern.

Das Programm verwendet einen 16-Bit-Zähler in zwei Nullseiten-Speicherplätzen für die Zählung der Zeit zwischen den Temperatur-Messungen.

```
;
;NAME DES PROGRAMMS: THERMOMETER
;DATUM: 5/1/79
;PROGRAMMIERER: LANCE. A. LEVENTHAL
;PROGRAMM-SPEICHER-ERFORDERNISSE: 173 BYTES
;BENÖTIGTES RAM: 5 BYTES
;E/A-ERFORDERNISSE: 1 EINGANGSPORT, 1 AUSGANGSPORT (1 VIA 6522)
;
;DIESES PROGRAMM STELLT EIN DIGITALES THERMOMETER DAR, DAS
;  EINGANGSSIGNALE VON EINEM A/D-WANDLER ANNIMMT, DER MIT
;  EINEM THERMISTOR VERBUNDEN IST. ES WANDELT DAS
;  EINGANGSSIGNAL IN GRAD CELSIUS UM UND ZEIGT DIE ERGEBNISSE
;  AUF ZWEI SIEBEN-SEGMENT-LED-ANZEIGEN AN.
;
;A/D-WANDLER
;
;DER A/D-WANDLER IST EIN MONOLITHISCHER KONVERTER 7570J VON
;  ANALOG DEVICES, DER EIN 8-BIT-AUSGANGSSIGNAL LIEFERT
;DER UMWANDLUNGSVORGANG WIRD DURCH EINEN IMPULS AUF DER
;  LEITUNG "START CONVERSION" (STEUERLEITUNG 2 AUF DEM
;  VIA-PORT B GESTARTET
;DIE UMWANDLUNG WIRD INNERHALB VON 50 MIKROSEKUNDEN
;  AUSGEFÜHRT, UND DIE DIGITALEN DATEN WERDEN
;  ZWISCHENGESPEICHERT
;
;ANZEIGEN
;
;ES WERDEN ZWEI SIEBEN-SEGMENT-LED-ANZEIGEN MIT SEPARATEN
;  DECODERN VERWENDET (7447 ODER 7448, ABHÄNGIG VON DER ART
;  DER ANZEIGE)
;DIE DATENEINGÄNGE DER DECODER SIND MIT DEN BITS 0 BIS 3 DES
;  VIA-PORTS B VERBUNDEN
;BIT 4 DES VIA-PORTS B WIRD ZUM AKTIVIEREN DER LED-ANZEIGEN
;  VERWENDET (BIT 4 IST 1 ZUM SENDEN VON DATEN ZU DEN LEDS)
;BIT 5 DES VIA-PORTS B DIENT ZUR AUSWAHL DER VERWENDETEN LED (BIT
;  5 IST 1, WENN DIE FÜHRENDE ANZEIGE VERWENDET WIRD, 0, WENN DIE
;  FOLGENDE ANZEIGE VERWENDET WIRD)
;
;VERFAHREN
;
;SCHRITT 1 - INITIALISIERUNG
;  DER SPEICHERSTAPEL (VERWENDET FÜR RÜCKKEHR-ADRESSEN UND
;  ANDERE SPEICHERZWECKE) WIRD INITIALISIERT
;SCHRITT 2 - PULSEN DER LEITUNG FÜR DEN START DER UMWANDLUNG
;  DIE LEITUNG FÜR DEN START DER UMWANDLUNG DES A/D-WANDLERS
;  WIRD GEPULST (STEUERLEITUNG 2 DES VIA-PORTS B)
;SCHRITT 3 - WARTEN AUF STABILEN ZUSTAND DES
;            A/D-AUSGANGSSIGNALS
;  DIE LEITUNG "BUSY" VOM WANDLER WIRD MIT DER STEUERLEITUNG 1
;  AM PORT A DES VIA VERBUNDEN. WENN "BUSY" AUF HIGH GEHT UM ZU
;  SIGNALISIEREN, DASS DIE SIGNAL-UMWANDLUNG ABGESCHLOSSEN IST,
;  SETZT SIE BIT 7 DES VIA-UNTERBRECHUNGSFLAG-REGISTERS.
```



```
;SCHRITT 4 - LESEN DES A/D-WANDLERS UND UMWANDLUNG IN GRAD
;            CELSIUS
;  FÜR DIE UMWANDLUNG WIRD EINE TABELLE VERWENDET, DIE DEN
;  MAXIMALEN EINGANGSWERT FÜR JEDE TEMPERATUR-ABLESUNG
;  ENTHÄLT.
;SCHRITT 5 - ANZEIGE DER TEMPERATUR AUF DEN LEDS
;  DIE TEMPERATUR WIRD AUF DEN LEDS SECHS SEKUNDEN LANG
;  ANGEZEIGT, BEVOR EINE WEITERE UMWANDLUNG AUSGEFÜHRT WIRD.
;
;
;
;DEFINITIONEN DER VARIABLEN DES THERMOMETERS
;
;
;SPEICHERSYSTEM-KONSTANTEN
;
BEGIN    =$0400          ;START-ADRESSE FÜR DAS PROGRAMM
STKBGN   =$FF            ;STARTADRESSE FÜR DEN STAPEL AUF
                         ;  SEITE 1
TEMP     =0              ;STARTADRESSE FÜR DEN RAM-SPEICHER
;
;E/A-EINHEITEN UND VIA-ADRESSEN
;
VIAORB   =$A000          ;AUSGANGSPORT FÜR ANZEIGEN
VIAORA   =$A001          ;EINGANGSPORT FÜR WANDLER
VIADDRB  =$A002          ;DATEN-RICHTUNGS-REGISTER FÜR PORT B
VIADDRA  =$A003          ;DATEN-RICHTUNGS-REGISTER FÜR
                         ;  PORT A
VIAPCR   =$A00C          ;PERIPHERES STEUERREGISTER DES VIA
VIAFIR   =$A00D          ;UNTERBRECHUNGSFLAG-REGISTER DES
                         ;  VIA
;
;ZEITWEILIGER RAM-SPEICHER
;
         *=TEMP
DCTRC    *=*+2           ;ANZEIGEIMPULS-ZÄHLER
INPUT    *=*+1           ;ZWISCHENSPEICHER FÜR
                         ;  WANDLER-EINGANG
LSTEMP   *=*+1           ;NIEDRIGSTWERTIGE STELLE DER
                         ;  TEMPERATUR
MSTEMP   *=*+1           ;HÖCHSTWERTIGE STELLE DER
                         ;  TEMPERATUR
;
;DEFINITIONEN
;
BUSYF    =%00000010      ;MUSTER ZUM PRÜFEN DES BUSY-STATUS
LEDON    =%00010000      ;CODE ZUM SENDEN DES AUSGANGS-
                         ;  SIGNALS ZU DEN LEDS
LEDSL    =%00100000      ;CODE ZUR AUSWAHL DER FÜHRENDEN
                         ;  ANZEIGE
MSCNT    =$C7            ;ERFORDERLICHE ZÄHLUNG FÜR 1 MS
                         ;  VERZÖGERUNGSZEIT
TSAMPH   =6              ;TSAMPL GIBT AN, WIE OFT DIE ANZEIGEN
                         ;  IN EINER ABTAST-PERIODE DER
                         ;  TEMPERATUR GEPULST WERDEN.
```

```
TSAMPL  =250                    ; DIE LÄNGE EINER ABTASTPERIODE IST
                                ; DAHER 2*TPULS*TSAMPH*TSAMPL-
                                ; MILLISEKUNDEN.DER FAKTOR VON
                                ; 2*TPULS WIRD DURCH DIE TATSACHE
                                ; BEDINGT, DASS JEDE DER ZWEI
                                ; ANZEIGEN TPULS MS LANG GEPULST
                                ; WIRD.
TPULS   =2                      ;ANZEIGE FÜR PULS-LÄNGE IN MS
;
;
;
        *=$FFFC
;
;RESET-ADRESSE DES THERMOMETER-PROGRAMMS
;
        WORD  BEGIN
;
;INITIALISIERUNG DES THERMOMETER-PROGRAMMS
;
        *=BEGIN
        LDX   #STKBGN           ;INITIALISIERE STAPELZEIGER
        TXS
        LDA   #0                ;MACHE PORT-A-LEITUNGEN ZU
                                ;  EINGÄNGEN
        STA   VIADDRA
        LDA   #$FF              ;MACHE PORT-B-LEITUNGEN ZU
                                ;  AUSGÄNGEN
        STA   VIADDRB
        LDA   #%11000001        ;MACHE "START CONVERSION" LOW,
                                ;  BUSY-AKTIV-LOW AUF HIGH
        STA   VIAPCR            ;KONFIGURIERE PERIPHERE STEUERUNG
                                ;  DES VIA
        LDA   #BUSYF            ;LÖSCHE BUSY-FLAG
        STA   VIAIFR
;
;PULSE LEITUNG FÜR DEN START DER UMWANDLUNG
;
START   LDA   #%11100001        ;SENDE START-UMWANDLUNG HIGH
        STA   VIAPCR
        LDA   #%11000001        ;SENDE START-UMWANDLUNG LOW
        STA   VIAPCR
;
;WARTE, BIS BUSY AUF HIGH GEHT UND LIES DATEN
;
        LDA   #BUSYF
WTBSY   BIT   VIAFR             ;IST UMWANDLUNG ABGESCHLOSSEN?
        BEQ   WTBSY             ;NEIN, WARTE
        LDA   VIAORA            ;JA, LIES DATEN VOM WANDLER
;
;WANDLE DATEN FÜR TEMPERATUR IN DEZIMAL UM
;
        JSR   CONVR             ;WANDLE DATEN FÜR TEMPERATUR IN
                                ;  BINÄR UM
        JSR   BINBCD            ;WANDLE BINÄR IN BCD UM
;
```



```
;KONFIGURIERE ZIFFERN FÜR ANZEIGE
;
        ORA   #LEDON            ;SETZE AUSGANG ZU LEDS (LSD IN A)
        STA   LSTEMP            ;BEWAHRE NIEDRIGSTWERTIGE STELLE
                                ;  AUF
        TXA                     ;HOLE HÖCHSTWERTIGE ZIFFER
        BEQ   SVMSD             ;LASSE ANZEIGE AUSGESCHALTET, WENN
                                ;  MSD NULL IST
        ORA   #LEDON            ;SETZE AUSGANG ZU LEDS
        ORA   #LEDSL            ;WÄHLE FÜHRENDE ANZEIGE AUS
SVMSD   STA   MSTEMP            ;BEWAHRE HÖCHSTWERTIGE STELLE AUF
;
;PULSE DIE LED-ANZEIGEN
;
PULSE   LDA   #TSAMPH           ;16-BIT-ZÄHLER FÜR ANZEIGE-IMPULSE
        STA   DCTR+1
TLOOP   LDA   #TSAMPL
        STA   DCTR
DSPLY   LDA   MSTEMP            ;AUSGABE ZU FÜHRENDER ANZEIGE
        STA   VIAORB
        LDY   #TPULS            ;VERZÖGERE ANZEIGE-IMPULSLÄNGE
        JSR   DELAY
        LDA   LSTEMP            ;AUSGABE ZU FOLGENDER ANZEIGE
        STA   VIAORB
        LDY   #TPULS            ;VERZÖGERE ANZEIGE-IMPULSLÄNGE
        JSR   DELAY
        DEC   DCTR
        BNE   DSPLY
        DEC   DCTR+1            ;HAT DIE ZÄHLUNG NULL ERREICHT?
        BNE   TLOOP             ;NEIN, SETZE PULSEN DER ANZEIGE FORT
        BEQ   START             ;JA, HOLE NEUEN TEMPERATURWERT
;
;UNTERPROGRAMM DELAY WARTET AUF DIE ANZAHL DER IM
;  INDEX-REGISTER Y DURCH ZÄHLUNG MIT DEM INDEX-REGISTER X
;  SPEZIFIZIERTEN MS
;
DELAY   LDX   #MSCNT            ;ZÄHLE FÜR 1 MS-VERZÖGERUNG
WTLP    DEX                     ;WARTE 1 MS
        BNE   WTLP
        DEY
        BNE   DELAY             ;ZÄHLE ANZAHL DER MS
        RTS
;
;UNTERPROGRAMM CONVR WANDELT EINGANGSSIGNAL VOM
;  A/D-WANDLER IN GRAD CELSIUS DURCH VERWENDUNG EINER TABELLE
;  UM. EINGANGSDATEN LIEGEN IM AKKUMULATOR UND ERGEBNIS IST
;  EINE BINÄRZIFFER IM AKKUMULATOR
;
```

```
;VERWENDETE REGISTER: A,X
;VERWENDETER SPEICHERPLATZ: EINGANG
;
CONVR   STA     INPUT       ;BEWAHRE GELESENES EINGANGSSIGNAL
                            ;  AUF
        LDX     #$FF        ;STARTE TABELLEN-INDEX BEI -1
CHVAL   INX                 ;INKREMENTIERE TABELLEN-INDEX
        LDA     DEGTB,X     ;HOLE EINGABE VON TABELLE
        CMP     INPUT       ;IST A/D-EINGANG UNTER EINGABE?
        BCC     CHVAL       ;NEIN, HALTE WEITER AUSSCHAU
        TXA                 ;JA, KEHRE MIT T IN AKKUMULATOR
                            ;  ZURÜCK
        RTS
;
;TABELLE DEGTB WURDE DURCH EICHUNG GEFUNDEN.
;DEGTB ENTHÄLT DIE GRÖSSTEN EINGANGSWERTE, DIE EINER SPEZIELLEN
;  TEMPERATUR-ABLESUNG ENTSPRECHEN (D.H., DIE ERSTE EINGABE IST
;  DEZIMAL 58, SO DASS EIN EINGANGSWERT VON 58 DER GRÖSSTE WERT
;  IST, DER EINE TEMPERATUR-ABLESUNG VON NULL ERGIBT - WERTE
;  UNTER NULL SIND NICHT GESTATTET)
;
DEGTB   .BYTE       58,61,63,66,69,71,74,77,80,84
        .BYTE       87,90,93,97,101,104,108
        .BYTE       112,116,120,124,128,132,136
        .BYTE       141,145,149,154,158,163,167
        .BYTE       172,177,181,186,191,195,200
        .BYTE       204,209,214,218,223,227,232
        .BYTE       236,241,245,249,253,255
;
;DAS UNTERPROGRAMM BINBCD WANDELT EINE BINÄRZAHL KLEINER ALS
;  100 IN ZWEI BCD-ZIFFERN UM. DIE EINGANGSDATEN LIEGEN IM
;  AKKUMULATOR, UND DIE ERGEBNISSE LIEGEN IM INDEX-REGISTER X
;  (HÖCHSTWERTIGE STELLE) UND IM AKKUMULATOR (NIEDRIGSTWERTIGE
;  STELLE)
;
;VERWENDETE REGISTER: A,X
;
BINBCD  LDX     #$FF        ;ZEHNER-ZÄHLUNG = -1
        SEC                 ;SETZE ÜBERTRAG ZU ANFANG
SUBTEN  INX                 ;INKREMENTIERE ZEHNER-ZÄHLUNG
        SBC     #10         ;KANN ZEHN NOCH SUBTRAHIERT
                            ;  WERDEN?
        BCS     SUBTEN      ;JA, SETZE PORT
        ADC     #10         ;NEIN, ADDIERE LETZTE ZEHN ZURÜCK
        RTS
        .END
```


LITERATUR

1. E. R. Hnatek, A User's Handbook of D/A and A/D Converters, Wiley, New York, 1976.

2. T. A. Seim, "Numerical Interpolation for Microprocessor-Based Systems," Computer Design, February 1978, pp. 111-116.

3. D. H. Sheingold, ed., Analog-Digital Conversion Notes, Analog Devices, Inc., P. O. Box 796, Norwood, MA. 02062, 1977.

4. D. P. Burton, and A. L. Dexter, Microprocessor Systems Handbook, Analog Devices, Inc., P. O. Box 796, Norwood, MA 02062, 1977.

5. J. B. Peatman, Microcomputer-based Design, McGraw-Hill, New York, 1977.

6. F. Molinari, et al., "Shopping for the Right Analog I/O Board", Electronic Design, October 11, 1978, pp. 238-243.

7. Auslander, D. M. et al., "Direct Digital Process Control: Practice and Algorithms for Microprocessor Applications", Proceedings of the IEEE, February 1978, pp. 199-208.

8. R. J. Bibbero, Microprocessors in Instruments and Control, Wiley, New York, 1977

9. A. Mrozowski, "Analog Output Chips Shrink A-D Conversion Software", Electronics, June 23, 1977, pp. 130-133.

10. P. R. Rony et al., "Microcomputer Interfacing: Sample and Hold Devices", Computer Design, December 1977, pp. 106-108.

11. P. H. Garrett, Analog Systems for Microprocessors and Minicomputers, Reston Publishing Co., Reston, VA., 1978.

12. The Optoelectronics Data Book, Texas Instruments, Inc., P. O. Box 5012, Dallas, TX., 1978.

12. The Optoelectronic Designer's Catalog, Hewlett-Packard, Inc., 1820 Embarcadero Road, Palo Alto, CA 94303, 1978.